Intel C/C++ Compiler User's Guide for Win32* Systems

Issued in U.S.A.
Order Number 664711-004

Intel C/C++ Compiler User's Guide for Win32 Systems*

Order Number: 664711-004

How to Use This Online Manual

Click to hide or show subtopics when the bookmarks are shown.

Double-click to jump to a topic when the bookmarks are shown.

Click to display bookmarks.

Click to display thumbnails.

Click to close bookmark or thumbnail view.

Click and use on the page to drag the page in vertical direction.

Click and drag on the page to magnify the view.

Click and drag on the page to reduce the view.

Click and drag the selection cursor to the page.

Click to go to the first page of the manual.

Click to go to the previous page.

Click to go to the next page.

Click to go to the last page.

Click to return to the previous view. Use this button when you need to go back after using the jump button (see below).

Click to go forward from the previous view.

Click to set 100% of the page view.

Click to display the entire page within the window.

Click to fill the width of the window.

Click to open a dialog to search for a word or multiple words.

Click jump button on manual pages to jump to the related subjects. Use the return back icon above to go back.

Printing an Online File: Select **Print** from the **File** menu to print an online file. The dialog that opens allows you to print full text, range of pages, or selection.

Viewing Multiple Online Manuals: Select **Open** from the **File** menu, and open the .PDF file you need. Select **Cascade** from the **Window** menu to view multiple files.

Resizing the Bookmark Area: Drag the double-headed arrow that appears on the area's border as you pass over it.

Jumping to Topics: In this manual, many of the function names and section titles are printed in green color, underlined style to indicate that you can jump to them.

On the next page, you will find a jump text demo.

Jump Text Demo

The following text segment is taken from the manual. In this example, the section name “Restricting Optimization” is displayed in green underlined font. If you click this section name, the first page of the section will display. To return to this page

click the  icon in the tool bar.

> This chapter describes ways to improve the performance of your application and the effects of optimization options on programs. The “Restricting Optimization” section tells you how to suspend optimizations for certain applications or for debugging.

Contents

Chapter 4 Optimizations

Chapter 7 C/C++ Language Features

Chapter 8 Microsoft Compatibility

Chapter 9 Diagnostic Information

Glossary

Index

Examples

Figures

Tables

About This Manual

This manual describes how to use the Intel C/C++ Compiler for Win32* systems. The Intel C/C++ Compiler can be hosted on either a Windows* NT* or Windows 95 operating system.

Chapters 1 through 3 help new users get started. Chapter 4 describes optimizations to fine-tune your application for maximum performance. The "Compiler Option Quick Guide" in Chapter 2 is an alphabetical list of compiler options with references to the full descriptions in the manual. Chapters 5 through 8 help you use preprocessing and maintain language conformance. Chapter 9 describes diagnostic information. Chapter 10 describes libraries. Chapter 11 describes how to control compiler-generated code. Chapter 12 lists MMX™ technology intrinsics. Appendix A provides a table of the compiler limits and Appendix B provides information on experimental performance tuning. This manual also provides a glossary and an index.

This manual does not teach you the C programming languages. As prerequisites, you should be familiar with the Intel processor architecture and be aware of the role of C and assembly-language programs in the software development process. You should also be familiar with the host computer's operating system.

The Intel C/C++ Compiler has been designed to operate as much like the Microsoft Visual C++* compiler as possible. In many cases, you can apply the functional descriptions provided with the Microsoft Visual C++

compiler to the Intel compiler, except where this manual states that you cannot. See "Application Development" in Chapter 1 for the relationship of the Intel C/C++ Compiler to system-specific programming support tools.

Related Publications

The following documents provide additional information relevant to the Intel C/C++ Compiler:

- *The Annotated C++ Reference Manual*, 1st edition, Ellis, Margaret; Stroustrup, Bjarne, Addison Wesley, 1991. Provides information on the C++ programming language.
- *The C Programming Language*, 2nd edition, Kernighan, Brian W.; Ritchie, Dennis W., Prentice Hall, 1988. Provides information on the K & R definition of the C language.
- *C: A Reference Manual*, 3rd edition, Harbison, Samual P.; Steele, Guy L., Prentice Hall, 1991. Provides information on the ANSI standard and extensions of the C language.
- For environment specifics, see the manuals or on-line help supplied with Microsoft Visual C++, 32-bit edition for Windows.
- For Win32 specific information, see the documentation included with the *Microsoft Win32 Software Development Kit, Version 3.1*.

Information about the target architecture is available from Intel and from most technical bookstores. Some helpful titles are:

- *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture,* Intel Corporation, order number 243190
- *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual,* Intel Corporation, order number 243190
- *Intel Processor Identification with the CPUID Instruction,* Intel Corporation, order number 241618
- *Intel Architecture MMX Technology Programmer's Reference Manual*, Intel Corporation, order number 241618.
- *Pentium® Pro Processor Developer's Manual (3 Volume Set)*. Intel Corporation, order number 242693.

- *Pentium Processor Specification Update*. Intel Corporation, order number 242480.
- *Pentium Processor Family Developer's Manual*, Intel Corporation, order numbers 241428-005.
- *Intel486™ Microprocessor Family Programmer's Reference Manual.* Intel Corporation, order number 240486.
- Most Intel documents are also available from the Intel Corporation web site at *www.intel.com*.

Notational Conventions

This manual uses the following conventions:

`This type style`	Indicates an element of syntax, a reserved word, a keyword, a filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
	`l` indicates lowercase letter L in examples. `1` is the number 1 in examples. `O` is the uppercase O in examples. `0` is the number 0 in examples.
`This type style`	Indicates the exact characters you type as input.
`This type style`	Indicates a place-holder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
`[`*`items`*`]`	Indicates that the items enclosed in brackets are options.
`{`*`item`*` | `*`item`*`}`	Indicates to elect only one of the items listed between braces. A vertical bar (|) separates the items.
`...` (ellipses)	Indicate that you can repeat the preceding item.

Overview

1

This chapter introduces you to the Intel C/C++ Compiler for Win32 Systems. The compiler is designed to operate on any 32-bit Intel Architecture machine. However, it produces optimized code for the Intel486, Pentium, and the Pentium Pro processors.

Tools You Need

To compile programs with this compiler you need the following tools:

- A Windows NT or Windows 95 operating system
- Microsoft Visual C++ Versions 4.0, 4.1, or 4.2 available on the host operating system

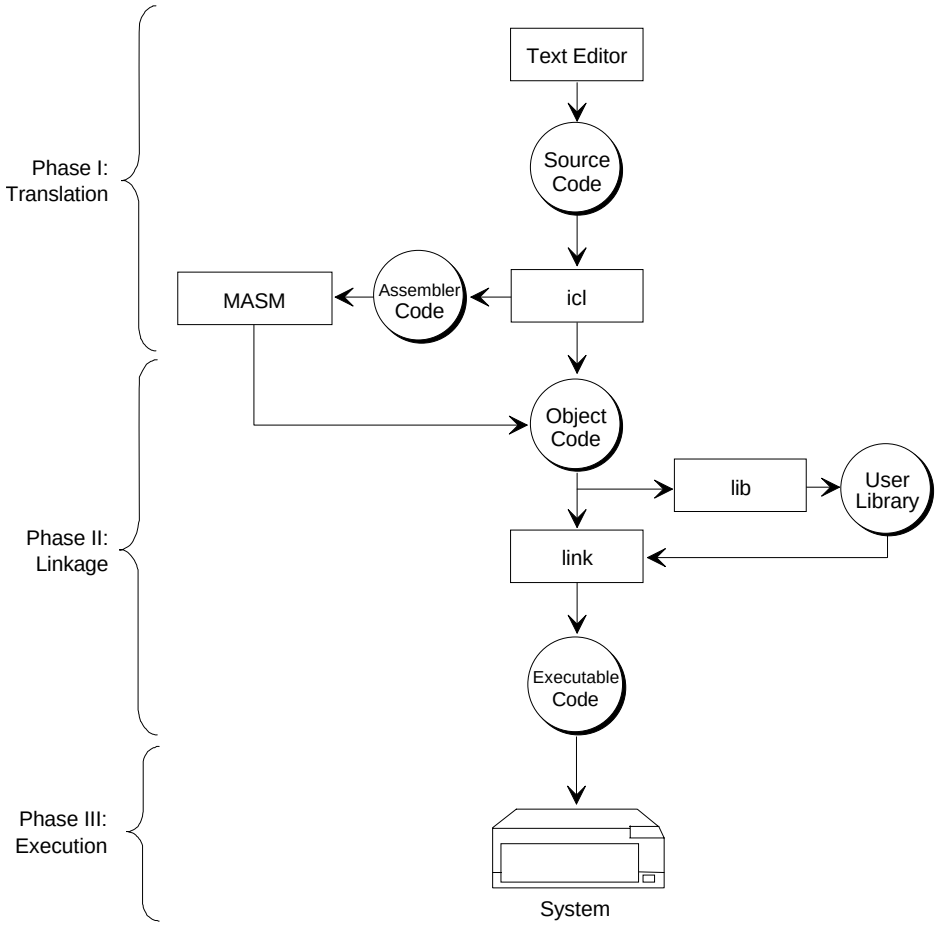
NOTE. *The Intel C/C++ Compiler is a plug-in product that uses the Microsoft Visual C++ header files, libraries, and many other components.*

If you plan to assemble files generated by the assembler or use the `-Quse_asm` option, you must provide an assembler. Version 6.11 of the Microsoft Macro Assembler (MASM) is recommended. Whatever assembler you use, you must name it `MASM.EXE` and place it in your default path.

Application Development

Figure 1-1 shows you how the Intel compiler (icl) fits into the Microsoft application development environment. The compiler processes C or C++ language source and generates either assembly or object modules. You decide the input and output by setting options when you run the compiler.

Figure 1-1 Application Development Cycle



Compiler Operation 2

This chapter describes how to run the compiler, how to select compilation options, and the default behavior of the compiler.

This chapter also contains the "Compiler Option Quick Guide," which is a list of all options the Intel C/C++ Compiler will accept. The options that you invoke change the compiler's default behavior in the following areas:

- controlling compilation flow
- controlling compiler output
- preprocessing files
- controlling optimization
- specifying language conformance
- preparing for debugging
- controlling the linker
- managing special cases
- preparing for profiling

The options available for the Intel C/C++ Compiler are described in detail in Chapters 3 through 11.

Compiler Command-line Syntax

To invoke the compiler from the DOS command line, enter the following command:

```
prompt> icl [options] [path]filenames
```

options	Indicates one or more command-line options. The compiler recognizes one or more letters preceded by a hyphen (-) or a forward slash (/) as an option.
filenames	Indicates one or more source files to be processed by the compilation system. You can specify more than one [*path*]*filename*. You must separate multiple filenames with a space.

NOTE. *You cannot associate a filename with an option by command-line placement alone. Specified options apply to all files. For example, in the following command line, the* `-Za` *option applies to both* `x.i` *and* `y.cpp`*:*

prompt> **`icl -c x.i -Za y.cpp`**

Using the Intel C/C++ Compiler from the Microsoft Visual C++ Development Environment

If you have Microsoft Visual C++ v. 4.0 on your system when you install the Intel C/C++ Compiler, the installation procedure automatically integrates the Intel C/C++ Compiler add-on with the Visual C++ development environment. This gives you the choice of using the Intel C/C++ Compiler to compile the performance-critical portions of the projects that you create in Visual C++.

To invoke the Intel C/C++ Compiler from within the Visual C++ development environment, follow these steps:

1. Open a Microsoft project (`.mdp` or `.mak`) file in Visual C++.
2. Click on the Tools menu in the Developer Studio.
3. Click Select Compiler. The Select Compiler window opens.
4. Click on the Intel C/C++ Compiler option.
5. Click OK.

For more information on using the Intel C/C++ Compiler, click the Help button in the Visual C++ development environment.

Filename Extensions

The compiler recognizes files with the following extensions as C++ files: `.cc`, `.cpp`, and `.cxx`. For consistency, to illustrate C++ examples, this manual uses the `.cpp` extension.

The compiler recognizes files with the extension `.c` as being C files, unless you specify individual C++ files with the `-Tp` *`file`* option.

In addition, the Intel C/C++ Compiler recognizes the default filename extensions listed in Table 2-1.

Table 2-1 Default Filename Extensions

Filename	Interpretation	Action
filename.lib	object library	Passed to LINK.exe.
filename.i	C or C++ source preprocessed and expanded by the C/C++ preprocessor	Passed to compiler.
filename.obj	compiled object module	Passed to LINK.exe.
filename.asm	assembly file	Assembled by MASM.

Compiler Option Quick Guide

Table 2-2 provides you with a reference of all compilation control options and some options for control of the linker.

Table 2-2 Summary of Command-line Options

Option	Description	Default	Reference
-?, -help	Prints compiler options summary.	OFF	page 9-1
-c	Stops the compilation process after an object file has been generated. The compiler generates an object file for each C or C++ source file or preprocessed source file.	OFF	page 5-4
-C	Places comments in preprocessed source output.	OFF	page 6-2
-D*name*[=*value*]	Defines a macro name and associates it with the specified value.	OFF	page 6-3
-E	Stops the compilation process after the C or C++ source files have been preprocessed, and writes the results to `stdout`.	OFF	page 6-2
-EP	Stops the compilation process after the C or C++ source files have been preprocessed and writes the results to `stdout`. The `#line` directives are stripped (that is, not included in the output).	OFF	page 6-2
-F *n*	Sets the amount of stack to reserve for the program (that is, passes `-stack:`*n* to the linker).	OFF	*
-FA[cs]	Produces an assembly output file. The arguments `c` and `s` have no effect on the Intel C Compiler.	ON	*
-Fa[*filename*]	Produces an assembly output file with the specified filename, or the default name if the filename is not specified.	OFF	page 5-4

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-Fe[*filename*]	Produces an executable output file with the specified filename, or the default name if the filename is not specified.	ON	page 5-4
-Fm[*filename*]	Instructs the linker to produce a map file.	ON	*
-Fo[*filename*]	Produces an object output file with the specified filename, or the default name if the filename is not specified.	ON	page 5-4
-G3	Targets the optimizations to all Intel Architecture processors.	ON	page 4-3
-G4	Targets the optimizations to the Intel i486 processor.	OFF	page 4-3
-G5	Targets the optimizations to the Intel Pentium processor.	OFF	page 4-3
-G6	Targets the optimizations to the Intel Pentium Pro and Pentium II processors.	OFF	page 4-3
-GB	Targets the optimizations to all Intel Architecture processors.	ON	page 4-3
-Ge	Enables stack-checking.	ON	*
-Gf	Enables string-pooling optimization.	ON	*
-GF	Enables read-only string-pooling.	ON	*
-Gh	Adds a call to user-supplied `_ _penter` routine in each function prolog.	OFF	*
-GR	Enables C++ RTTI.	OFF	*
-GR-	Disables C++ RTTI.	ON	*
-Gr	Select Microsoft _fastcall as the default calling convention.	OFF	*
-Gs *n*	Enables stack-checking for routines with *n* or more bytes of local variables and compiler temporaries.	OFF	*
-Gy	Enables function-level linking.	ON	*
-GX	Enables C++ exception handling.	OFF	page 7-6

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-GX-	Disables C++ exception handling.	ON	page 7-6
-H *n*	Limits the length of external symbol names to *n* characters.	OFF	*
-I*directory*	Specifies an additional directory to search for include files whose names do not begin with a slash (/).	OFF	page 3-5
-J	Makes default `char` type unsigned.	OFF	*
-LD	Produces a DLL.	OFF	*
-LDd	Create Debug version of DLL.	OFF	*
-link	Passes options to the linker.	OFF	page 3-5
-MD	Compiles and links with the dynamic, multi-thread C run-time library.	OFF	*
-MDd	Compiles and links with the dynamic, multi-thread, debug version of the C run-time library.	OFF	*
-ML	Compiles and links with the static, single-thread C run-time library.	ON	*
-MLd	Compiles and links with the static, single-thread, debug version of the C run-time library.	OFF	*
-MT	Compiles and links with the static, multi-thread C run-time library.	OFF	*
-MTd	Compiles and links with the static, multi-thread, debug version of the C run-time library.	OFF	*
-O1	Optimize for speed, but disable some optimizations that increase code size for small speed benefit. The -O1 option has the same effect as specifying the following options: -Og, -Oi-, -Os, -Oy, -Ob1, -Gf, -Gs, and -Gy.	OFF	page 4-1

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-O2	Optimizes for speed. The -O2 option has the same effect as specifying the following options:-Og, -Oi, -Ot, -Oy, -Ob1, -Gf, -Gs, and -Gy.	ON	page 4-1
-Od	Disables optimizations.	OFF	page 4-1
-Og	Enables global optimizations.	ON	*
-Oi	Enables in-line expansion of standard library functions.	ON	page 4-8
-Oi-	Disables in-line expansion of standard library functions.	OFF	page 4-8
-Op	Specifies conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic. Optimization is restricted accordingly.	OFF	page 4-9
-Op-	Favors optimization over conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic.	ON	page 4-9
-Os	Enables most speed optimizations, but disable optimizations that increase code size for a small speed benefit.	OFF	*
-Ot	Enables all speed optimizations.	ON	*
-Ox	Same as the -O2 option without the -Gf and -Gy options.	OFF	*
-Oy	Enables the use of the ebp register in optimizations.	ON	page 5-5
-Oy-	Disables the use of the ebp register in optimizations. Instead, it is used as the frame pointer.	OFF	page 5-5
-P	Stops the compilation process after C or C++ source files have been preprocessed and writes the results to files named according to the compiler's default file-naming conventions.	OFF	page 6-2

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-QA-	Causes all predefined macros (other than those beginning with _ _) and assertions to be inactive. (Same as -u.)	OFF	page 6-3
-QA*name*[(*values*)]	Associates a symbol name with the specified sequence of values; equivalent to an `#assert` preprocessing directive.	OFF	page 6-3
-QH	Produces a list of include file dependencies, one per line, to `stdout`.	OFF	page 6-4
-QI0f	Avoids the incorrect decoding of certain 0f instructions.	OFF	page 10-4
-QIfdiv	Enables a software patch for the floating-point division flaw that exists in some steppings of the Pentium processor.	OFF	page 10-3
-QIfdiv-	Disables the software patch for the floating-point division flaw that exists in some steppings of the Pentium processor.	ON	page 10-3
-Qip	Enables interprocedural optimizations.	OFF	page 4-5
-Qipo	Enables interprocedural optimization across files.	OFF	page 4-4
-Ob*<n>*	Controls the compiler's inline expansion. The amount of inline expansion performed varies with the value of *<n>* as described in the options that follow.		page 4-5
-Ob0	Disables inlining.		page 4-5
-Ob1	Enables inlining of functions declared with the _inline keyword. Also enables inlining according to the C++ language.		page 4-5
-Ob2	Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the -Qip option.		page 4-5
-QM	Generates makefile dependency lines for each source file, based on the `#include` lines found in the source file.	OFF	page 6-5

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-Qmem	Specifies memory optimizations that improve cache hits and reduce the number of memory accesses.	OFF	page 4-7
-Qnobss_init	Places variables that are initialized with zeroes in the DATA section.	OFF	page 11-2
-Qpc *nn*	Enables floating-point significand precision control. The value of *nn* is used to round the significand to the correct number of bits. The value of *nn* should be either 32, 64, or 80.	OFF	page 4-10
-Qprec	Improves floating-point precision with less speed impact than -Op.	OFF	page 4-10
-Qprof_dir <*dirname*>	Specifies the directory to hold profile information.	OFF	page 4-15
-Qprof_gen	Instruments the program to collect basic block execution counts.	OFF	page 4-15
-Qprof_genx	Generates an instrumented object file and also creates a new static profile information file (`.spi`).	OFF	page 4-15
-Qprof_use	Uses dynamic feedback information.	OFF	page 4-15
-Qrcd	Disables changing of the FPU rounding control.	OFF	page 4-11
-Qrct	Sets internal FPU rounding control to truncate.	OFF	page 4-11
-Qscope_alloc	Minimizes stack space allocation by overlapping variables from disjoint scopes.	OFF	page 11-2
-Quse_asm	Compiles source files to assembly files and calls MASM to generate object files from the assembly files.	OFF	page 5-4
-QW*tool*,*arguments*	Passes an argument list to another program in the compilation sequence, such as the assembler or linker.	OFF	page 3-4
-Qwd*tag*	Disables the soft diagnostic that corresponds to *tag*.	OFF	page 9-5

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-Qwe*tag*	Overrides the severity of the soft diagnostic corresponding to *tag* and makes it an error.	OFF	page 9-5
-Qwn*num*	Limits the number of errors displayed prior to aborting compilation to *num*.	100	page 9-6
-Qwr*tag*	Overrides the severity of the soft diagnostic corresponding to *tag* and makes it a remark.	OFF	page 9-5
-Qww*tag*	Overrides the severity of the soft diagnostic corresponding to *tag* and makes it a warning.	OFF	page 9-5
-Qxi	Enables generation of instructions for the Pentium Pro and Pentium II processors.	OFF	page 4-4
-S	Stops the compilation process after an assembly source has been generated. The compiler writes assembly code source for each C or C++ source file or preprocessed source file to a file using the output file's naming conventions.	OFF	page 5-2
-TC	Compile all source or unrecognized file types as C source files.	OFF	*
-Tc *file*	Treats *file* as a C source file.	OFF	*
-Tp *file*	Treats *file* as a C++ source file.	OFF	*
-TP	Compile all source or unrecognized file types as C++ source files.	OFF	*
-u	Causes all predefined macros (other than those beginning with _ _) and assertions to be inactive. (Same as -QA-.)	OFF	*
-U*name*	Suppresses any definition of a macro name; equivalent to a `#undef` preprocessing directive.	OFF	page 6-3
-V *text*	Sets version string to *text*.	OFF	*

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-vd0	Suppresses C++ hidden `vtordisp` constructor/destructor displacement member.	OFF	*
-vd1	Generates C++ hidden `vtordisp` constructor/destructor displacement member.	ON	*
-vmb	Selects best-case (smallest) representation for pointers to members. Use this option if each class is always defined before any pointer to a member of the class is declared.	ON	*
-vmg	Selects general representation for pointers to members. Use this if a pointer to a class member is declared prior to the definition of the corresponding class.	OFF	*
-vmm	Used with -vmg, allows pointers to members of single- and multiple-inheritance classes.	OFF	*
-vms	Used with -vmg, allows pointers to members of single-inheritance classes.	OFF	*
-vmv	Used with -vmg, allows pointers to members of classes of any inheritance type.	OFF	*
-W0	Specifies that only error messages are displayed (no warnings or remarks).	OFF	page 9-4
-W1, -W2, -W3	Specifies that error and warning messages are displayed.	ON	page 9-4
-W4	Specifies that error, warning, and remark messages are displayed.	OFF	page 9-4
-X	Removes the standard directories from the list of directories to be searched for include files.	OFF	page 3-4

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Table 2-2 Summary of Command-line Options (continued)

Option	Description	Default	Reference
-Za	Enforces strict conformance to the ANSI standard for C.	OFF	page 7-2
-Zd	Produces only line number information in the object file (for debugging).	OFF	*
-Ze	Accepts extended C language.	ON	page 7-2
-Zi	Generates symbolic debugging information in the object code for use by source-level debuggers.	OFF	page 5-5
-Zl	Disables embedding default libraries in object files.	OFF	*
-Zp*number*	Specifies the strictest alignment constraint for structure and union types as one of the following: 1, 2, 4, 8, or 16 bytes.	8	page 11-1
-Zs	Checks the syntax of a program and stops the compilation process after the C or C++ source files and preprocessed source files have been parsed. Generates no code and produces no output files. Warnings and messages appear on `stderr`.	OFF	page 5-2

* *The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information about these options, see the documentation that accompanied your copy of Microsoft Visual C++.*

Default Behavior of the Compiler

You need not enter any options when you invoke the Intel C/C++ compiler. If you do not specify any options, the compiler uses default settings. The compiler driver performs the following actions by default:

- Produces executable output with the filename of the first source or object file on the command line with a `.exe` suffix.
- Searches for include files using the `INCLUDE` variable.
- Searches for library files in directories specified by the `LIB` variable, if they are not found in the current directory.
- Sets 8 bytes as the strictest alignment constraint for structures.
- Displays error and warning messages.
- Uses ANSI with extensions (`-Ze`) for C source files.
- Performs standard optimizations using the default `-O2` option, as described in "Optimization Choices" in Chapter 4.

If the compiler does not recognize a command-line option, that option is ignored and a warning is displayed. See Chapter 9, "Diagnostic Information" for detailed descriptions about system messages.

Changing the Compilation Environment 3

This chapter describes how to customize the environment you use during compilation. Through customizing you can specify the following:

- names and locations of compilation tools
- options to include during each compilation
- options and files to use during individual projects
- directories in which to search for include files and libraries

Environment Variables

In the `LIB` and `PATH` environment variables, you must place the names of directories that contain the math libraries and the compiler executable files respectively. In the `INCLUDE` environment variable, place the names of the directories in which the compiler will search for include files.

The compiler uses the directory specified by the `TMP` variable to store temporary files it creates. If the directory specified by `TMP` does not exist, the compiler places the temporary files in the current directory.

Refer to your operating system documentation to find out how to specify values for the environment variables.

Configuration Files

A user configuration file named `icl.cfg` can be added to the directory where `icl.exe` is installed. In this file, you can specify common options that you want to include in each compilation. Options in the configuration file are processed in the order they appear, followed by the command-line options that you specify when you invoke the compiler.

You can decrease the time you spend entering command-line options and ensure consistency by using the configuration file to automate some command-line entries. However, you should be aware that options placed in the configuration file will be included each time you run the compiler. If you have varying option requirements for different projects, see "Response Files" later in this chapter.

You can insert any valid command-line option into this file. Use the pound "`#`" character to treat the rest of the line as a comment.

Example 3-1 illustrates a sample `icl.cfg` file.

Example 3-1 Sample icl.cfg File

```
## Sample icl.cfg file.
##
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
##
## Additional directories to be searched for include
## files, before the default.
-Ic:\project\include
##
## Use the static, multi-threaded C run-time library.
-MT
```

Response Files

You can use response files to specify options used during particular compilations and save this information in individual files. Response files are invoked as an option on the command line. Options in a response file are inserted in the command line at the point where the response file is invoked.

You can decrease the time you spend entering command line options and ensure consistency by using response files to automate command line entries. Further, you can use individual response files to maintain options for specific projects and avoid editing the configuration file when changing from one project to the next.

Any number of options or filenames can be placed on a line in the response file. You can also use several response files in the same command line. You can insert any valid command-line option into this file. Use the pound "#" character to treat the rest of the line as a comment.

The syntax for using response files is as follows:

```
prompt> cl @response_file filenames
```

Note that an "at" sign (@) must precede the name of the response file on the command line.

Tools

The compiler calls `LINK.exe`, the Microsoft linker, to produce an executable file from the object files. This linker searches the path specified in the `LIB` environment variable to find the libraries.

The Microsoft Visual C++ package does not contain an assembler. To use the `-Quse_asm` option, you must provide an assembler. By default, the compiler expects to find the `MASM.exe` assembler in your path. If you have a 32-bit version of an assembler, you can rename it `MASM.exe` and place it in your `PATH` environment variable.

Include Files

By default, the compiler searches for the standard include files in the directories specified in the `INCLUDE` environment variable. You can also indicate the location of include files in the configuration file, `icl.cfg`, using the options described in "Specifying an Include Directory (-I)".

Specifying an Include Directory (-I)

By default, the compiler searches for include files in the directory specified by the INCLUDE environment variable. Use the -I*directory* option to specify an additional directory in which to search for include files. Included files are brought into the program with a #include preprocessor directive. The compiler searches directories for include files in the following order:

- directory of the source file that contains the include
- directories specified by the -I option
- directories specified in the INCLUDE environment variable

Removing Include Directories (-X)

Use the -X option to prevent the compiler from searching the default path specified by the INCLUDE environment variable.

You can use the -X option with the -I option to prevent the compiler from searching the default path for include files and direct it to use an alternate path. For example, to direct the compiler to search the path \alt\include instead of the default path, use the following command line:

```
prompt> icl -X -I \alt\include newmain.cpp
```

Passing Options to Other Programs (-QW*tool*)

Use the -QW option to pass an argument to a compilation tool used later in the development sequence. For example, some options for the assembler (masm) or linker (link) programs are not recognized by icl. The -QW option takes two arguments in the following format:

```
-QWtool, arguments, [arguments...]
```

tool Indicates one of the following letters that designate the compilation tool to receive one or more of these arguments:

- p Specifies the compiler front-end program and C (or C++) preprocessor.

	`0` or `2`	Specifies the compiler back-end program. For the Intel C/C++ Compiler the `0` and `2` options are interchangeable. Use `-QW0` (or `-QW2`) to fine-tune optimizations, as described in "Keywords for Optimization (-QW0)" in Appendix B.
	`a`	Specifies the assembler (`masm.exe`).
	`l`	Specifies the linker (`link.exe`).
`arguments`		Indicates one or more valid arguments for the designated program. If the argument is a command-line option, you must include the hyphen. If the argument contains a space or tab character, you must enclose the entire argument in quotation characters (`" "`). You can separate multiple arguments with commas.

The following example directs the linker to create a memory map when the compiler produces the executable file from the source.

```
prompt> icl -QWl,-map:proto.map proto.cpp
```

The `-QWl` option in the preceding example is passing the **`-map`** option to the linker. This is an explicit way to pass arguments to other tools in the compilation process.

However, to get the same results implicitly you could also use the following command:

```
prompt> icl -Fmproto.map proto.cpp
```

Passing Options to the Linker (-link)

In addition to using the `-QWl` option to pass options to the linker, you can use the `-link` option. The compiler passes all options following `-link` to the linker. For example, to compile the file `a.cpp` and instruct the linker to link `a.obj` with `libfoo.lib`, creating `a.exe`, type the following at the command line:

```
prompt> icl a.cpp -link libfoo.lib
```

Optimizations 4

This chapter describes ways to improve the performance of your application and the effects of optimization options on programs. The "Restricting Optimization" section tells you how to suspend optimizations for certain applications or for debugging. You can find related information in Appendix B, "Experimental Performance Tuning," which describes some experimental ways to tune your code. The "Glossary" provides definitions of all optimizations mentioned in this chapter.

Optimization Choices

To specify the kind of optimization you want for your program, use the following command-line options:

`-Od`	Disables optimizations.
`-O1`	Enables options `-Og`, `-Oi-`, `-Os`, `-Oy`, `-Ob1`, `-Gf`, `-Gs`, and `-Gy`. However, `-O1` disables a few options; specifically, those that increase code size for a small speed benefit such as in-line expansion of library functions. This option is not as aggressive as the Microsoft Visual C++ Compiler version of the `-O1` option. In most cases, `-O2` is recommended over `-O1` because the `-O2` option enables in-line expansion, which helps programs that have many function calls.

`-O2`	Enables options `-Og`, `-Oi`, `-Ot`, `-Oy`, `-Ob1`, `-Gf`, `-Gs`, and `-Gy`. Confines optimizations to the procedural level. See Table 4-1 for a complete list of the kinds of optimizations performed. The `-O2` option is the default optimization option.
`-Qip`	Enables optimizations that cross procedural boundaries.
`-Qipo`	Enables interprocedural optimization between files.
`-Qmem`	Concentrates on improved memory use.

Table 4-1 shows the optimizations that the compiler applies to your program for each optimization option. The entry "any" in the Option column means that the compiler automatically performs this optimization, even when optimizations are disabled (using the `-Od` option).

Table 4-1 Optimization Summary

Optimization	Affected Aspect of Program	Option
optimized code selection	instruction selection/ addressing modes	any
global register allocation	register use	-O1 / -O2
instruction scheduling	instruction reordering	-O1 / -O2
register variable detection	register use	-O1 / -O2
common subexpression elimination	constants and expression evaluation	-O1 / -O2
dead-code elimination	instruction sequencing	-O1 / -O2
variable renaming	register use	-O1 / -O2
loop-invariant code movement	instruction sequencing	-O1 / -O2
copy propagation	constants and expression evaluation	-O1 / -O2
constant propagation	constants and expression evaluation	-O1 / -O2
strength reduction/induction variable simplification	instruction selection/sequencing constants and expression evaluation	-O1 / -O2
tail recursion elimination	calls, further optimization	-O1 / -O2
loop unrolling	instruction selection/sequencing	-Qmem
loop interchange	memory access	-Qmem
in-line function expansion	calls, jumps, branches, and loops	-Qip

continued ☛

Table 4-1 Optimization Summary (continued)

Optimization	Affected Aspect of Program	Option
interprocedural constant propagation	arguments, global variables, and return values	-Qip
passing arguments in registers	calls, register usage	-Qip
monitoring module-level static variables	further optimizations, loop invariant code	-Qip
multifile optimization	affects the same aspects as -Qip, but across multiple files	-Qipo

Restricting Optimization

As described under "Optimization Choices" the `-Od` option disables optimizations.

Targeting a Processor (-G*n*)

By default, the compiler produces binary programs for a variety of Intel processors. These programs run on any Intel Architecture 32-bit processor. However, they may run at lower performance levels on specific processors. If you target a binary application to run exclusively on a specific processor and you want maximum performance, use the `-Gn` option to optimize for that processor. Regardless of which of the `-Gn` suboptions you choose, the resulting binary will run on any Intel Architecture 32-bit processor. The suboptions for `-Gn` indicate the type of processor on which the binary will run. The suboptions have the following descriptions:

`GB` or `G3` — Choose `GB` or `G3` when the binary application must run on a variety of Intel processors. This type blends optimization for performance across the entire Intel family of processors. The `GB` and `G3` options are ON by default.

`G4` — Choose `G4` when the binary application runs exclusively on the Intel486 processor.

`G5` — Choose `G5` when the binary application runs exclusively on the Pentium processor.

G6	Choose G6 when the binary application runs on the Pentium Pro or Pentium II processors.

For example, the following command compiles the source program prog.cpp and optimizes for the Pentium Pro processor:

```
prompt> icl -G6 prog.cpp
```

Generating Pentium Pro and Pentium II Instructions (-Qxi)

Use the -Qxi option to generate instructions for the Pentium Pro and Pentium II processors. Use the -Qxi option only in combination with the -G6 option. If you direct the compiler to generate instructions for the Pentium Pro processor, your program will run only on the Pentium Pro processor (that is, it will not run on the Pentium or any previous-generation Intel processors). The following command compiles the source program prog.cpp, and optimizes and generates instructions for the Pentium Pro processor:

```
prompt> icl -G6 -Qxi prog.cpp
```

Interprocedural Optimization (IPO) with -Qip and -Qipo

The compiler performs the following interprocedural optimizations (IPOs):

- Inline function expansion
- Interprocedural constant propagation

These optimizations are most effective for a program that contains many small or medium-sized frequently used functions. Programs that contain calls within loops can be optimized by using -Qip or -Qipo.

NOTE. *The -Qip and -Qipo options in some cases can significantly increase compile time. This increase is due to the interprocedural analyses and inlining. Also, -Qipo enables whole-program compilation, which increases link-time compilation. Further, the inline function expansions performed by -Qip and -Qipo might increase code size, which can slow down some programs.*

Single File IPO (-Qip)

For function calls, the compiler can replace the original instructions with more efficient instructions to reduce execution time. See Table 4-1 for a complete list of the `-Qip` optimizations. The primary `-Qip` optimization that performs such restructuring is in-line function expansion. The following section describes how the compiler performs in-line function expansion.

In-line Function Expansion

For function calls, the compiler can replace the original instruction containing a call to a function with the function body itself. Using a call to a function within a program takes less space but requires longer execution time than repeating the function body at each point where the function is called. This substitution, also called in-lining, benefits your application as follows:

- It eliminates the overhead of saving the registers and pushing the arguments and the return address required to make a function call.
- It allows the compiler to perform optimizations involving statements that call a function.
- It brings information about the called function into the calling function and, conversely, makes information about the calling function accessible within the called function.

Such expansion increases execution speed, but the program size also increases.

Certain criteria must be met before the compiler can perform in-line function expansion. See "Criteria for In-Line Function Expansion" in Appendix B for more information on these criteria.

Multifile IPO (-Qipo)

The goal of multifile IPO is to obtain potential optimization information from individual program modules of a multifile program. For example, if the compiler finds a call to a function that can be optimized through in-line function expansion, it can perform the in-line expansion even if the function is in another module. The result of using `-Qipo` is an optimized executable compiled from multiple files.

See "Profile-Guided Optimization" for a description of how you can make further use of multifile optimizations so that other optimizations can benefit from profile information.

Creating a Multifile IPO Executable

To achieve multifile interprocedural optimization, do the following:

1. Compile your programs with the `-Qipo` option. For example, in the following command lines, the `a.cpp`, `b.cpp`, and `c.cpp` source modules are compiled with the `-Qipo` option:

   ```
   prompt> icl -G6 -Qipo -c a.cpp
   prompt> icl -G6 -Qipo -c b.cpp
   prompt> icl -G6 -Qipo -c c.cpp
   ```

   The `-c` option stops the compilation process after an object file has been generated. When you compile with `-Qipo` you save an intermediate form of each module in a file parallel to the object file with a `.il` suffix. The name and location of this file are the same as that of the object file.

   The resulting object files for `a.cpp`, `b.cpp`, and `c.cpp` are `a.obj`, `b.obj`, and `c.obj`. The intermediate form and summary information for the modules is stored in `a.il`, `b.il`, and `c.il`.

2. With the preceding example, the following command optimizes interprocedurally across the modules listed on the command line:

   ```
   prompt> icl -Fefilename -G6 -Qipo a.obj b.obj c.obj
   ```

   The `-Fe` option is used to store the executable in *`filename`*.

If a `.il` file does not exist, the object file is passed on to the final link stage. Multifile interprocedural optimizations still occurs among modules for which information is available in up-to-date `.il` files.

The commands in steps 1 and 2 can be combined and replaced with the following command:

```
prompt> icl -G6 -Qipo -Fefilename a.cpp b.cpp c.cpp
```

NOTE. *If you move an object file to a different directory, you must move the `.il` file to the same directory.*

Multifile IPO and Linker Arguments

For a `-Qipo` command line that results in linkage of your program, the mixed order of object files and other linker arguments (if they exist) is not preserved. Instead, the linker arguments that you specify follow the `-Qipo` optimized object and any other objects you specified in the linker's command line.

Creating a Multifile IPO Executable Using a Project Makefile

Multifile interprocedural optimization involves an additional compilation step immediately before the link step. This step is hidden when you compile and link using `icl`. However, most applications are built using a makefile or a file with a similar function to a makefile. Makefiles usually call the Microsoft linker `link.exe` to do the linking.

However, when you use the `-Qipo` option to accomplish multifile interprocedural optimization you must use the Intel linker driver `xilink.exe` instead. The following is the generic command-line usage for `xilink.exe`:

```
prompt> xilink [-Qipo] <LINK_commandline>
```

where `-Qipo` is used to enable multifile interprocedural optimization and `<LINK_commandline>` is your usual linker command line. Do not use `-Qipo` when you are not performing multifile interprocedural optimizations.

Use the `xilink.exe` syntax when you use a makefile instead of step 2 in the example "Creating a Multifile IPO Executable". For example, you could type something like the following:

```
prompt> xilink -Qipo -out:filename.exe a.obj b.obj c.obj
```

Optimizing Memory and Cache Hits (-Qmem)

Several optimizations improve cache usage, which results in a reduced number of memory accesses. Memory-usage optimizations are most effective for scientific or engineering types of programs, especially programs that execute intensive matrix calculations. You activate these

memory usage optimizations with the **-Qmem** option. If you use **-Qmem**, you should also enable the interprocedural optimizations (**-Qip** or **-Qipo**) to obtain the best optimization.

The compiler performs memory optimizations only on code that references data in regular patterns, such as **for** loops that access data with constant bounds. The memory usage optimizations that you activate by using the **-Qmem** option affect the way the compiler performs the following tasks:

- Transform loops can access memory in consecutive locations.
- Applications can reuse the contents of the data cache.
- The compiler avoids external bus turn-around penalties by preloading useful data.
- The compiler unrolls loops to improve execution speed.

NOTE. *The* **-Qmem** *option can significantly increase compile time. This increase is due to the extra analyses performed by the memory optimizer. In addition, the overhead from loop transformations performed by* **-Qmem** *can increase code size. This increased size can also slow down some programs.*

In-line Expansion of Library Functions (-Oi, -Oi-)

By default, the compiler automatically expands a number of standard C, C++, and math library functions at the point of the call to that function, which usually results in faster computation.

The `errno` variable is not set when the functions are expanded in-line. Library routines, instead of their corresponding intrinsics expanded in-line, are used when you specify any of the following options at compilation: IEEE 754 conformance (`-Op`), ANSI standard conformance (`-Za`), restrict optimization (`-Od`), and debugging information (`-Zi`). Your results can vary slightly under these circumstances.

The compiler cannot always determine the difference between a library routine and another routine with the same name. To be sure the intended routine is used, use the `-Oi-` option whenever your program contains user-supplied routines with the same names as the standard routines.

This option does not interfere with the in-line expansion of functions in your program during interprocedural analysis. For example, the following command compiles the program `sum.cpp` without expanding the library functions:

```
prompt> icl -Qip -Oi- sum.cpp
```

NOTE. *Automatic in-line expansion of library functions is not related to the in-line expansion that the compiler does during interprocedural optimizations.*

Floating-point Arithmetic Precision (-Op, -Op-, -Qprec, -Qpc, -Qlong_double)

The options described in this section all provide optimizations with varying degrees of precision in floating-point arithmetic.

When to Use -Op

The `-Op` option restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI and IEEE standards.

For most programs, specifying this option adversely affects performance. If you are not sure whether your application needs this option, try compiling and running your program both with and without it to evaluate the effects on performance versus precision.

Specifying this option has the following effects on program compilation:

- User variables declared as floating-point types are not assigned to registers.
- Whenever an expression is spilled, it is spilled as 80 bits (extended precision), not 64 bits (double precision).
- Floating-point arithmetic comparisons conform to IEEE 754.

- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.
- The compiler performs floating-point operations in the order specified without reassociation.
- The compiler does not perform the constant-folding optimization. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point arithmetic is not performed to ensure that floating-point exceptions are also maintained.
- Floating-point operations conform to ANSI C. When assignments to type `float` and `double` are made, the precision is rounded from 80 bits (extended) down to 32 bits (`float`) or 64 bits (`double`). When you do not specify `-Op`, the extra bits of precision are not always rounded before the variable is reused.
- The `-Oi-` option, which disables inline functions expansion, is used.

NOTE. *The `-Oi-` and `-Op` options are active by default when you choose the `-Za` (strict ANSI C conformance) option.*

When to Use -Qprec

Use the `-Qprec` option to improve floating-point precision. The `-Qprec` option disables fewer optimizations and has less impact on performance than `-Op`.

When to Use -Qpc

Use the `-Qpc` *nn* option to enable floating-point significand precision control. Some floating-point algorithms, created for specific 32- and 64-bit systems, are sensitive to the accuracy of the significand or fractional part of the floating-point value. Set *nn* to one of the following values to round the significand to the indicated number of bits:

`-Qpc 32`	24 bits (single precision)
`-Qpc 64`	53 bits (double precision)

`-Qpc 80` 64 bits (extended precision)

The default value for *nn* is 64 for full floating-point precision.

This option allows full optimization. Using this option does not have the negative performance impact of using the `-Op` option because only the fractional part of the floating-point value is affected. The range of the exponent is not affected.

The `-Qpc` *nn* option causes the compiler to change the floating point precision control when the `main()` function is compiled. The program that uses `-Qpc` *nn* must use `main()` as its entry point, and the file containing `main()` must be compiled with `-Qpc` *nn*..

When to Use -Qlong_double

Use the `-Qlong_double` to change the size of the `long double` type to 80-bits. For compatibility with the Microsoft compiler, the Intel compiler's default `long double` type is 64 bits in size, the same as the `double` type. This option introduces a number of incompatibilities with other files compiled without this option and with calls to library routines. Therefore, Intel recommends that the use of `long double` variables be local to a single file when you compile with this option.

Rounding Control Options (-Qrcd and -Qrct)

The Intel compiler uses the `-Qrct` and `-Qrcd` options to improve the performance of code that requires floating point calculations. The optimization is obtained by controlling the change of the rounding mode.

The system default floating point rounding mode is round-to-nearest. This means that values are rounded during floating point calculations. However, the C language requires floating point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating point conversion and change it back afterwards. With `-Qrct` and `-Qrcd`, you can optimize your code by eliminating the instructions required to change the rounding mode back and forth. Naturally, changing the rounding mode causes small differences in the values obtained in floating point calculations.

The `-Qrcd` option disables the change to truncation of the rounding mode in floating point-to-integer conversions. This means that all floating point calculations must use the default round-to-nearest, including floating point-to-integer conversions. This option has no effect on floating point calculations, but conversions to integer will not conform to C semantics.

The `-Qrct` option causes the floating point rounding mode to always truncate when the `main()` function is compiled. Because the mode is always set to truncation, `-Qrct` disables the *change* to truncation. This means that all calculations are truncated, including all floating point calculations. This option can lead to small differences in floating point results, due to floating point calculations being truncated instead of rounded. The program that uses `-Qrct` must use `main()` as its entry point, and the file containing `main()` must be compiled with `-Qrct`.

Table 4-2 Summary of -Qrcd and -Qrct

Option	Floating Point-to-integer Conversions	Floating Point Calculations	Must Compile in `main()` Function?
-Qrcd	faster (not using C semantics)	Standard (round)	No
-Qrct	faster (using C semantics)	Non-Standard (truncate)	Yes

Alternatives to the -Qpc nn and -Qrct Options

The `-Qpc nn` and `-Qrct` options perform their intended function if your program has a `main()` routine and only if you use the Intel compiler to compile the file that contains `main()`. Otherwise, these options are not useful.

An alternative method to achieve the effect of `-Qpc` *nn* or `-Qrct`, which does not require the use of the Intel compiler to compile `main()`, is through the use of the Microsoft C library routine `_controlfp()`. The `_controlfp()` function allows you to modify the floating-point precision control or rounding control. To use this routine, you must include the Microsoft header file `float.h`.

Table 4-3 summarizes the way to achieve the effects of `-Qpc nn` or `-Qrct` using `_controlfp()`.

Table 4-3 -Qpc and -Qrct Equivalent Calls

Option	Equivalent Call	Extra Options
-Qpc 32	_controlfp(_PC_24, _MCW_PC);	none
-Qpc 64	_controlfp(_PC_53, _MCW_PC);	none
-Qpc 80	_controlfp(_PC_64, _MCW_PC);	none
-Qrct	_controlfp(_RC_CHOP, _MCW_RC);	-Qrcd

NOTE. *It is preferable to insert the call to* `_controlfp()` *within an initialization routine in your program.*

To achieve the effects of `-Qrct`, you must add the `_controlfp()` and compile your program with the `-Qrcd` option.

Figure 4-1, below, provides an example of how `_controlfp()` might be used in a program. Please refer to the Microsoft On-line Help for more information on the `_controlfp()` function.

Figure 4-1 Example of the _controlp() Function:

```
/* init.c - contains initialization code for my application
 */

#include <float.h>

void init_program(void)
{
  // Set FPU precision control to use 24-bit significand
  _controlfp( _PC_24, _MCW_PC );

  // other initialization code follows...
 }
```

Profile-Guided Optimization

Profile-guided optimization consists of three sequential phases:

- instrumentation compilation
- instrumented execution
- feedback compilation

This section defines each of these phases, describes the profile-guided options to use in each phase, and then provides an example. A description of each of the three phases follows:

1. **Instrumentation compilation:** When you compile your program, the compiler inserts code into your program to produce profile information. The resulting code is said to be instrumented by the compiler.
2. **Instrumented execution:** When you execute the instrumented program with your data sets, it creates a dynamic information file (a file with a `.dyn` extension). The data in these files represents the actual behavior of the program during execution; this provides an accurate picture of how the program runs with a particular set of data. By default, these files are placed in the directory in which the program was compiled, regardless of where the instrumented program is executed.
3. **Feedback compilation:** When you compile your program a second time, the compiler uses the data in the dynamic information files to help optimize your program. The data helps the compiler determine the most heavily traveled paths through the program and optimizes along these paths. You might want to use additional optimization options during feedback compilation so that other compiler optimization routines can benefit from the dynamic information. As part of the feedback compilation, the dynamic information files are merged into a summary file called `pgopti.dpi`. This file is created in the same directory as the dynamic information files.

When to Use Profile-Guided Optimization

When you use profile-guided optimizations, consider the following usage guidelines:

- Use profile-guided optimizations on performance-critical areas of large applications.
- Minimize the changes you make to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for files that have been modified after that information was generated. The compiler issues a warning when this occurs. The compiler still uses the dynamic information corresponding to unmodified source files.
- Repeat the instrumentation compilation if you make significant changes to your source files after instrumented execution and before feedback compilation.

Table 4-4, which follows, describes the options to use for profile-guided optimizations. Some of these options are demonstrated in the example that follows Table 4-4.

Table 4-4 Profile-Guided Optimization Options

Option	Description
`-Qprof_gen`	Specifies instrumentation compilation and instructs the compiler to produce instrumented code in preparation for instrumented execution. If your program is built in multiple directories, use the `-Qprof_dir` option for both the instrumentation compilation (`-Qprof_gen`) and the feedback compilation (`-Qprof_use`) phases.
`-Qprof_genx`	Generates an instrumented object file and also creates a new static profile information file (`.spi`). When you run an instrumented program using `-Qprof_genx` instead of `-Qprof_gen`, you are able to use the `proforder` tool to create a function order list for the linker. However, using `-Qprof_genx` for instrumentation requires more memory at runtime, and it produces larger `.dyn` files. In such a case `-Qprof_gen` is sufficient for performance tuning. Also, you cannot execute parallel make files when you use `-Qprof_genx`

continued ☛

Table 4-4 Profile-Guided Optimization Options (continued)

Option	Description
`-Qprof_dir <dirname>`	Specifies the directory where dynamic information files are created.The default directory is the directory in which the program is compiled. The specified directory must already exist. You should specify the same `-Qprof_dir` option for both the instrumentation and feedback compilations. However, if you move the dynamic information files, you need to specify the new path.
`-Qprof_use`	Specifies feedback compilation and instructs the compiler to use available dynamic information to produce a profile-optimized executable. You can use the `-Qip` or `-Qmem` options so that other compiler optimization routines can benefit from the dynamic information. If you perform multiple executions of the instrumented program, `-Qprof_use` merges the dynamic information files (`.dyn`) again and overwrites the old `pgopti.dpi` file.

Table 4-5 describes environment values that you can set to determine the directory in which to store dynamic information files or whether to overwrite the `pgopti.dpi` file. Refer to your operating system documentation for instructions on how to specify environment values.

Table 4-5 Profile-Guided Optimization Environment Variables

Variable	Description
`PROF_DIR`	Specifies the directory in which dynamic information files are created.The value of this variable, if specified, overrides the location specified by the `-Qprof_dir` option and the default location. This variable applies to all three phases of the profiling process.
`PROF_NO_CLOBBER`	This environment variable alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges the data from all dynamic information files and creates a new `pgopti.dpi` file, even if one already exists. When this variable is set, the compiler does not overwrite the existing `pgopti.dpi` file. In this case, the compiler issues a warning and you must remove the `pgopti.dpi` file if you want to use additional dynamic information files.

Using Profile-Guided Optimization: An Example

The following is an example of how to use profile-guided optimizations. Assume your program consists of three source files: `a1.cpp`, `a2.cpp`, and `a3.cpp`.

1. **Instrumentation compilation:** Compile the source files with the `-Qprof_gen` option and link the resulting object files to produce the instrumented program `a1.exe` as follows:

   ```
   prompt> icl -Qprof_gen -c a1.cpp a2.cpp a3.cpp
   prompt> icl a1.obj a2.obj a3.obj
   ```

   In place of the second command, you could use the linker (`link.exe`) directly to produce the instrumented program.

2. **Instrumented execution:** Run your instrumented program `a1.exe` with a representative set of data (if the program requires data):

   ```
   prompt> a1.exe
   ```

   You can run the program more than once with different input data. The instrumented program generates a dynamic information file with a unique name and `.dyn` suffix each time `a1.exe` is executed.

3. **Feedback compilation:** Compile and link the source files with the `-Qprof_use` option to use the dynamic information:

   ```
   prompt> icl -Qprof_use -Qip a1.cpp a2.cpp a3.cpp
   ```

   The compiler produces the `pgopti.dpi` file in the directory in which you performed the instrumented compilation in phase 1.

You can use different compilation options for phases 1 and 3. This example used the compiler default optimizations (`-O2`) in phase 1 and the interprocedural optimizations (`-Qip`) in phase 3.

NOTE. *The compiler does not allow you to use the* `-Qip` *or the* `-Qipo` *option with* `-Qprof_gen`*. Also, you cannot use* `-Qmem` *with* `-Qprof_gen`*.*

Profile Guided Optimizations Using a Function Order List

Profile-guided optimizations can support the generation of a function order list to be used by the Microsoft Visual C++ Linker. A function order list is a text file that specifies the order in which the linker should link the non-static functions of your program. The order is determined by the compiler using profile information. A function order list improves the performance of your program by reducing paging and improving code locality.

To enable the Intel C/C++ compiler and `proforder` tool to generate a function order list, you must use the `-Qprof_genx` option instead of the `-Qprof_gen` option during instrumentation compilation. When you use `-Qprof_genx`, the Intel C/C++ Compiler generates an extra output file, `pgopti.spi`, in the current directory or the directory specified by `-Qprof_dir`.

You will need to use the utilities `profmerge` and `proforder` described later in this chapter in "Utilities for Profile-Guided Optimization".

Function Order List Usage Guidelines

The following are usage guidelines that you must follow to properly use a function order list.

1. The order list only affects the order of non-static functions.
2. Do not use the `-Qprof_genx` option to compile two files of the same program simultaneously. This means that you cannot use the `-Qprof_genx` option with parallel `makefile` utilities.
3. To use the function order list, you must compile with the `-Gy` option. This option is active when you specify either option `-O1` or `-O2`.

Function Order List Example

Assume you have a C program that consists of files `file1.c` and `file2.c` and that you have created a directory for the profile data files in `c:\profdata`. The following are the steps for generating and using a function order list.

1. Compile your program with profile instrumentation by specifying the `-Qprof_genx` option as well as the `-Qprof_dir` option:

```
prompt> icl -FeMYPROG -Qprof_genx -Qprof_dir
c:\profdata file1.c file2.c
```

2. Run the instrumented program on one or more sets of input data.

   ```
   prompt> MYPROG.exe
   ```

   The program produces a `.dyn` file each time it is executed.

3. Merge the data from one or more runs of the instrumented program using the `profmerge` tool to produce the `pgopti.dpi` file.

   ```
   prompt> profmerge -Qprof_dir c:\profdata
   ```

4. Generate the function order list using the `proforder` tool. By default, the function order list is produced in the file `proford.txt`.

   ```
   prompt> proforder -Qprof_dir c:\profdata
   -o MYPROG.txt
   ```

5. Compile your application with profile feedback by specifying the `-Qprof_use` and the `/ORDER` option to the linker. Again, use the `-Qprof_dir` option to specify the location of the profile files.

   ```
   prompt> icl -FeMYPROG -Qprof_use -Qprof_dir
   c:\profdata file1.c file2.c -link /ORDER:@MYPROG.txt
   ```

Utilities for Profile-Guided Optimization

You can use the `profmerge` and `proforder` utilities to help you merge dynamic information files and improve the performance of your programs. Descriptions of these utilities follow.

The profmerge Utility

Use this tool after the instrumented execution phase to merge the dynamic profile information files (`.dyn`). The compiler executes this tool automatically during the feedback compilation phase when you specify the `-Qprof_use` option.

You might need to run this tool if you are generating a function order list for use with the `/ORDER` option provided by the Microsoft Visual C++ linker. See the `proforder` utility description, which follows, for more information.

The command-line usage for `profmerge` is as follows:

```
prompt> profmerge [-nologo] [-prof_dir <dir_name>]
```

This merges all .dyn files in the current directory, or the directory specified by -prof_dir, and produces the summary file pgopti.dpi. This file is used during feedback compilation with -Qprof_use and by the proforder tool.

The proforder Utility

The proforder tool is used as part of the feedback compilation phase, in conjunction with the Microsoft Visual C++ linker, to improve the performance of your program. The tool generates a function order list for use with the /ORDER option provided by the linker. The command line syntax for this tool is as follows:

```
prompt> proforder [-nologo] [-prof_dir <dir_name>]
[-o <order_file>]
```

where <dir_name> is the directory containing the profile files (.dpi, .dyn, and .spi) and <order_file> is the optional name of the function order list file to be generated. The default name, proford.txt, is generated if you do not specify a name.

Function Call to Dump Profile Data Explicitly

As part of the instrumented execution phase of profile-guided optimization, the instrumented program writes profile data to the dynamic information file (.dyn file). The file is written after the instrumented program returns normally from main() or calls the standard C exit function. For programs that do not terminate normally, the _PGOPTI_Prof_Dump function is provided. During the instrumentation compilation (-Qprof_gen) you can add a call to this function to your program. You should add the following function prototype prior to the call:

```
void _cdecl _PGOPTI_Prof_Dump(void);
```

NOTE. *You must remove the call prior to the feedback compilation (-Qprof_use).*

Specifying Compilation Output

5

This chapter describes all the Intel C/C++ options that determine the output created by the compiler in a variety of ways. By default, the compiler converts source code directly to an executable file. However, with certain options, you can control the output file that is produced by the compiler.

This means that you can create a file at any of the compilation phases such as assembly, object, or executable. If no errors occur during processing, you can use the output files from a particular phase as input to a later compiler invocation. Table 5-1 describes the options you can use to control the output.

Table 5-1 Compiler Input and Output Summary (continued)

Last Phase Completed	Limiting Option	Compiler Input	Compiler Output
Preprocessing	-P, -E, or -EP	source files	preprocessed files (See "Preprocessing Only (-E, -EP, and -P)" in Chapter 6 for more information about these options.)
Syntax checking	-Zs	C or C++ source files preprocessed files	diagnostic list
Compilation	-S	source files preprocessed files	assembly-language files

continued ☛

Table 5-1 Compiler Input and Output Summary (continued)

Last Phase Completed	Limiting Option	Compiler Input	Compiler Output
Assembly	-c	source files preprocessed files assembly-language files	unlinked object files
	-Quse_asm	source files	assembly files first, then object files
Linking	(default)	source files preprocessed files assembly files object files libraries	executable file map file linkable object file
compilation, linking, or assembly	-Fa, -Fo, -Fe	source, assembly, or object files	assigns a name to an output file of your choosing

Parsing for Syntax Only (-Zs)

Use the `-Zs` option to stop processing source files after they have been parsed for C/C++ language errors. This option gives you a way to check quickly whether sources are syntactically and semantically correct. The compiler creates no output file. In the following example, the compiler checks a file named `prog1.cpp`.

Any diagnostics appear on the standard error output.

```
prompt> icl -Zs prog1.cpp
```

Producing an Assembly Code File (-S)

Use the `-S` option to generate assembly files with the `.asm` suffix. The compiler does not perform the assembly and linking phases after the source files have been preprocessed and compiled. The file contains comments indicating the source-code line that corresponds to each assembly language instruction. Use the `-Fa` option to name the resulting file. By default, the compiler uses the name of each source file with the `.asm` extension.

For example, the following command creates two assembly language files, `file.asm` and `file2.asm`, that can later be assembled and linked:

```
prompt> icl -S file.cpp file2.cpp
```

CAUTION. *Any existing file with the same name and extension is overwritten.*

The following is an example of a portion of an assembly file listing:

```
_B1_3:; 11; preds: B1_2 B1_3
    addeax, 4; 7
    jne_B1_3; PROB 95%; 8
                    ; LOE: eax, ebx, esi, edi, esp
```

- `_B1_3` identifies the beginning of the third basic block in the first function of the file. A basic block is a set of instructions with the property that if the first instruction is executed then all of the subsequent instructions in the set are also executed.
- `; n` following the basic block label is the block execution count. This count is only printed when the `-Qprof_use` option is used. It indicates the average number of times a block was executed when the instrumented program was run. See "Profile-Guided Optimization" in Chapter 4 for more information on `-Qprof_use`.
- `; Preds` is a list of predecessors of the current basic block. These are blocks that can transfer control to the current basic block.
- The number following the semicolon (`;`) at the end of each instruction indicates the source line number corresponding to that assembly language instruction.
- `; Prob 95%` indicates the probability assigned to a conditional jump.
- `; LOE` indicates a list of registers which are live on exit from the current basic block. These are registers thatcontain values to be used by succeeding basic blocks.

Suppressing Linking (-c)

Use the `-c` option to suppress linking. For example, entering the following command produces the object files `file.obj` and `file2.obj`:

```
prompt> icl -c file.cpp file2.cpp
```

NOTE. *The preceding command does not link these files to produce an executable file.*

Using the Microsoft Assembler to Produce Object Code (-Quse_asm)

Use the `-Quse_asm` option to generate assembly code from input source files and then call the Microsoft assembler (MASM) to generate the object files. By default, the compiler generates an object file directly without going through the assembler.

For example, the following command generates assembly code, then calls the assembler to generate object files:

```
prompt> icl -c -Quse_asm file1.cpp
```

Linking

If you do not specify any of the preceding phase limitation options, the compiler defaults to creating executable files; providing that your code is error-free. The following sections describe options that you can use to change the name of the output file and how to prepare for debugging.

Naming the Output File (-Fe, -Fo, -Fa)

When compiling and linking a set of source files, you can use the `-Fe`, `-Fo`, or `-Fa` options to give the resulting file a name other than that of the first source or object file on the command line. If you are processing a single file, you can use the `-Fe`*`name`*, `-Fo`*`name`*, and `-Fa`*`name`* options to specify alternate names respectively for executable, object, and assembly files. Do

not enter a space between the option and the *name* argument. You can also use these options to override the default filename extensions .asm and .obj. In the following example, the -Fo option assigns the name file.obj to an output object file rather than the default (x.obj). The -c option directs the compiler to suppress linking.

```
prompt> icl -c -Fofile.obj x.cpp
```

In the following example, the command produces an executable file named outfile.exe as the result of compiling and linking two source files.

```
prompt> icl -Feoutfile.exe file.cpp file2.cpp
```

When compiling one or more files, the -Fe, -Fa, and -Fo options can be given a *dirname* (that is, a directory name) argument. The *dirname* argument must end in a forward slash "/" or backslash "\" character, and it must name an existing directory. In this case, the compiler uses the default convention in naming the executable, assembly, or object files produced, but the files will be placed in the directory specified by *dirname*.

In the following example, assume that obj_dir is an existing directory. The -Fe option causes the compiler to create the executable myprog.exe in the current directory. The -Fo option causes the compiler to create the object files a.obj, b.obj, and c.obj and place them in the directory obj_dir.

```
prompt> icl -Femyprog.exe -Foobj_dir/ a.cpp b.cpp c.cpp
```

Do not enter a space between the option and the *dirname* argument. You can specify different name arguments for each of the -Fe, -Fa, and -Fo options. The compiler does not remove objects that it produces, even when the compilation proceeds to the link phase.

Preparing for Debugging (-Zi, -Oy, -Oy-)

Use the -Zi option to direct the compiler to generate code to support symbolic debugging. For example:

```
prompt> icl -Zi prog1.cpp
```

The compiler uses `-Od` for the default optimization setting when the `-Zi` option is specified. Specifying the `-Zi` or `-Od` option automatically disables the `-Oy` option. See the description of the `-Oy-` option in the following section.

The compiler does not support the generation of debugging information in assembly files. If you specify the `-Zi` option with `-Quse_asm`, the resulting object file will not contain debugging information. If you specify the `-Zi` option with `-S` and later assemble the resulting assembly file, the resulting object file will not contain debugging information. If you specify the `-Zi` option with `-Fa`, the resulting object file will contain debugging information, but the assembly file will not.

The `-Oy-` option, as its name implies, disables the `-Oy` option. The `-Oy` option, which is enabled by default or when `-O1` or `-O2` is specified, allows the compiler to use the `ebp` register as a general purpose register in optimizations. However, most debuggers expect `ebp` to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-Oy-` option instructs the compiler to generate code that maintains and uses `ebp` as a stack frame pointer, without turning off optimization, so that a debugger can still produce a stack backtrace. Using this option reduces the number of available general purpose registers by one, and can result in slightly less efficient code.

Optimizations and Support for Symbolic Debugging

As described above, specifying the `-Zi` or `-Od` option automatically disables the `-Oy` option. The compiler lets you generate code to support symbolic debugging while the `-Qmem`, `-O1`, or `-O2` optimization options are specified on the command line along with `-Zi`. However, you can receive these unexpected results:

- If you specify the `-O1`, `-O2`, or `-Qmem` options with the `-Zi` option, some of the debugging information returned may be inaccurate as a side-effect of optimization.
- If you specify the `-O1`, `-O2`, or `-Qmem` options, the `-Oy` option will not be disabled. In this case, if you want to maintain the frame pointer while generating debug information, you must explicitly specify the `-Oy-` option on the command line to disable `-Oy`.

If you specify the `-Qip` optimization option with the `-Zi` option, the compiler issues a warning and ignores the `-Qip` option.

The effects of using the `-Zi` option with the optimization options are summarized in Table 5-2.

Table 5-2 Effects of Using -Zi with Optimization Options (continued)

These options	Imply these results
-Zi	debugging information produced, -Od, -Oy disabled
-Zi -O2	debugging information produced, -O2 optimizations enabled
-Zi -O2 -Oy-	debugging information produced, -O2 optimizations enabled, -Oy disabled
-Zi -Qmem	debugging information produced, -O2 and -Qmem optimizations enabled
-Zi -Qmem -Oy-	debugging information produced, -O2 and -Qmem optimizations enabled, -Oy disabled
-Zi -Qip	same results as -Zi -O2 (-Qip option ignored)

See the respective sections in Chapter 4, Optimizations for detailed descriptions of the optimizations options listed in Table 5-2.

Preprocessing 6

This chapter describes the options you can use from the command line to direct the operations of the preprocessor. Preprocessing performs such tasks as macro substitution, conditional compilation, and file inclusion. Table 6-1 provides a summary of the preprocessing options.

Table 6-1 Options to Control Preprocessing

Option	Description
-QA*name*[(*tokens*)]	Associates a symbol name with the specified sequence of values; equivalent to an `#assert` preprocessing directive.
-QA-, -u	Causes all predefined macros (other than those beginning with _ _) and assertions to be inactive.
-C	Preserves comments in preprocessed source output.
-D*name*[=*value*]	Defines the macro *name* and associates it with the specified *value*. The default (-D*name*) defines a macro with a *value* of 1.
-E	Directs the preprocessor to expand your source module and write the result to standard output.
-EP	Same as -E but does not include `#line` directives in the output.
-P	Directs the preprocessor to expand your source module and store the result in a file in the current directory.
-U*name*	Suppresses any automatic definition for the specified macro.

Preserving Comments in Preprocessed Source Output (-C)

Use the `-C` option to preserve comments in your preprocessed source output.

Preprocessing Only (-E, -EP, and -P)

Use either the `-E` or the `-P` option to preprocess your source files without compiling them.

When you specify the `-E` option, the compiler's preprocessor expands your source module and writes the result to standard output. The preprocessed source contains `#line` directives, which the compiler uses to determine the source file and line number during its next pass. For example, to preprocess two source files and write them to `stdout`, enter the following command:

```
prompt> icl -E prog1.cpp prog2.cpp
```

When you specify the `-P` option, the preprocessor expands your source module and stores the result in a file in the current directory. There is no way to change the default name. The preprocessor uses the name of each source file with the `.i` extension. For example, the following command creates two files named `prog1.i` and `prog2.i`, which you can use as input to another compilation:

```
prompt> icl -P prog1.cpp prog2.cpp
```

The `-EP` option can be used in combination with `-E` or `-P`. It directs the preprocessor to not include `#line` directives in the output. Specifying `-EP` alone is the same as specifying `-E -EP`.

CAUTION. *When you use the `-P` option, any existing files with the same name and extension are overwritten.*

Defining Macros (-QA, -QA-, -u, -D, and -U)

You can use the -QA and -D options to define the assertion and macro names to be used during preprocessing. The -U option directs the preprocessor to suppress an automatic definition of a macro.

Use the -QA option to make an assertion. This option performs the same function as the #assert preprocessor directive. The form of this option is:

-QAname[(value)]

name	Indicates an identifier for the assertion.
value	Indicates a value for the assertion. If a value is specified, it should be quoted, along with the parentheses delimiting it.

For example, to make an assertion for the identifier fruit with the value orange,banana, use the following command:

```
prompt> icl -QA"fruit(orange,banana)" prog1.cpp
```

The compiler provides a number of predefined macros. For a list of predefined macros available to the Intel C/C++ Compiler , see "Predefined Macros" below.

Enter -QA- or -u to suppress all predefined macros, except for those beginning with the double underscore.

Use the -D option to define a macro. This option performs the same function as the #define preprocessor directive. The form of this option is:

-Dname[=value]

name	The name of the macro to define.
value	Indicates a value to be substituted for *name*. If you do not enter a value, *name* is set to 1. The value should be quoted if it contains non-alphanumerics.

For example, to define a macro called SIZE with the value 100 use the following command:

```
prompt> icl -DSIZE=100 prog1.cpp
```

Use the -U*name* option to suppress any automatic definition for the specified name. The -U option performs the same function as a #undef preprocessor directive.

For more details about preprocessor directives, see a language reference such as *C: A Reference Manual*.

Predefined Macros

The predefined macros available for the Intel C/C++ Compiler are described in Table 6-2. The **Default** column describes whether the macro is enabled (ON) or disabled (OFF) by default. The **Disable** column lists the option that disables the macro; no indicates that the macro cannot be disabled.

Table 6-2 Predefined Macros

Macro Name	Default	Disable	Description / When Used
__ICL	ON	no	identifies the Intel C/C++ Compiler for Win32 systems
__cplusplus	C++ only	no	defined when compiling C++ source
_WIN32	ON	-u	defined for Win32 applications
_MSC_VER=1000	ON	-u	defined for Microsoft Visual C++ 4.0 compatibility
_M_IX86=*n*	ON, *n*=300	-u	defined based on the processor option you specify: *n*=300 if you specify the -GB or -G3 option *n*=400 if you specify the -G4 option *n*=500 if you specify the -G5 option *n*=600 if you specify the -G6 option
_DLL	OFF	-u	defined if you specify the -MD option
_MT	OFF	-u	defined if you specify the -MD, -MT, or -LD option
_CHAR_UNSIGNED	OFF	-u	defined if you specify the -J option
_CPPUNWIND	OFF	-u	defined if you specify the -GX option

See "Predefined Macros for ANSI Standard Conformance" in Chapter 7 for the list of predefined macros required for conformance to the ANSI standard.

Printing Include-file Dependencies (-QH)

Use the `-QH` option to print the pathname for each file compiled into the source with a `#include` directive. Pathnames can be absolute or relative. The compiler displays the dependencies on the standard output and prints the path for each included file on a separate line. The compilation process stops after preprocessing is completed.

In the following example, the source file, `dtest.cpp`, includes three other files. Enter the following command to display the dependent files:

```
prompt> icl -QH dtest.cpp
./d1.h
./d2.h
./d3.h
```

Printing Makefile Dependencies (-QM)

Use the `-QM` option to generate list of makefile dependency lines for each source file in the compilation. The compiler displays the dependency lines based on `#include` lines that appear in each source file. For example, the output of a simple program with one include file might be as follows:

```
hello.obj: hello.c
hello.obj: c:/Msdev/Include/stdio.h
```

C/C++ Language Features

7

This chapter describes the C and C++ language implementation of the Intel C/C++ Compiler. This chapter does not teach you how to program in C or C++. Instead, it covers the following topics:

- conformance to the ANSI standard for C
- options that support the C++ language

Conformance to C Standards

The compiler accepts two C language dialects: strict ANSI conformance, and extended ANSI. You can select the style that best suits your application.

The compiler's implementation of C conforms to the ANSI standard for the C language (X3.159-1989). The ANSI standard specifies that a conforming implementation of a C compiler must meet minimum requirements for certain translation limits. In all cases, the compiler exceeds these limits. Appendix A, "Compiler Limits," lists the tested values. The compiler also accepts ANSI C with extensions.

By default, the compiler does not flag extensions with error or warning messages. You must use the `-Za` option for the compiler to enforce the ANSI standard strictly. For more information on this option, see "Strict ANSI Dialect (-Za)" in this chapter.

C Language Dialects

The compiler supports two C language dialects: strict ANSI and ANSI with extensions. By default, the compiler accepts the language defined by the ANSI-with-extensions dialect. You can use the `-Za` option to select the language as defined by the ANSI standard [X3.159-1989] with no deviation, or the `-Ze` option to select the language defined by the ANSI standard with extensions.

The following sections describe each dialect. See the *C: A Reference Manual*, or *The C Programming Language*, all listed in the "About This Manual" section, for a full description of the C language.

Strict ANSI Dialect (-Za)

Use the `-Za` option to select the strict ANSI dialect. The compiler does not deviate from the ANSI standard when this option is in effect. While following strict conformance to the ANSI standard, the compiler flags any non-ANSI code in the source file with warnings or error messages.

Extended ANSI Dialect (-Ze)

When you enter the `-Ze` option on the invocation line, the compiler accepts the language specified in the ANSI standard with extensions in the areas described below.

Extensions for files and data storage:

- The input file can contain no text so you can use an empty file as input to the compiler.
- The last member of a structure can have an incomplete array type. However, that member cannot be the only member of the structure, otherwise, the structure size would be zero.
- A file-scope array can have an incomplete `struct` or `union` type as its element type. The `struct` or `union` type must be completed before the array is subscripted and by the end of the compilation if the array is not an `extern`.
- You can define an `enum` tag name then resolve it later in the source file.
- An initializer expression that is a single value and is used to initialize an entire static array, structure, or union need not be enclosed in braces. ANSI C requires the braces.

- The compiler generates a remark message for a storage-class specifier appearing anywhere other than the first position in a list of specifiers, as in `int static`.

Extensions for pointers:

- In an initializer, you can cast a pointer constant value to an integral type if the integral type is big enough to contain it.
- In an integral constant expression, you can cast an integer constant to a pointer type and then back to an integral type.
- The compiler accepts assignments of pointers to integers and to other incompatible pointer types without an explicit cast.
- You can select a field in the form `p->field`, even if `p` does not point to a `struct` or `union` that contains `field`. The `p` variable must be a pointer. Likewise, `x.field` is allowed even if `x` is not a structure or union that contains `field`. The `x` variable must be an `lvalue`. For both cases, all definitions of `field` as a field must have the same type offset within their structure or union.

Extensions for types and syntax:

- Bit fields can have `enum` base types, or integral types other than `int` and `unsigned int`.
- You can use the `long float` type as a synonym for `double`.
- You can place arbitrary text at the end of preprocessing directives.
- Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token. Under strict ANSI conformance, the compiler uses the `pp-number` syntax. See the ANSI standard text for more information on the `pp-number` syntax.

Extensions for predicates:

- You can use the `#assert` and `#unassert` macros to define and test predicate names.

Extensions with Warnings. When you use the `-Ze` option, the compiler accepts and tags the extensions in syntax and semantics as described below; however, these extensions are tagged with a warning message.

Extensions for syntax with warnings:

- You can have an extra comma at the end of an `enum` list.
- You can omit the final semicolon preceding the closing brace (`}`) of a structure or union.
- A right brace can immediately follow a label definition. Normally, a statement must follow a label definition.
- You can have an empty declaration (a semicolon with nothing before it).

Extensions for semantics with warnings:

- You are allowed assignment and pointer differences between pointers to types that are interchangeable but not identical, such as `unsigned char *` and `char *`. However, the compiler issues a warning. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- You can compare a pointer to a `void` to a pointer of another kind using the `>`, `>=`, `<`, or `<=` operators without using an explicit type cast. ANSI C allows such comparisons using `==` or `!=` and issues no warnings.
- You can insert in-line assembly code using the `asm` keyword. In the strict ANSI C dialect, such insertions are disabled.
- You can have free-standing tag declarations in the parameter declaration list for a function with old-style parameters.
- The compiler generates a warning if an overflow is detected while folding signed integer operations on constants.

Predefined Macros for ANSI Standard Conformance

The ANSI standard requires that certain predefined macros be supplied with the compiler. Table 7-1 lists the macros the compiler defines in accordance with the ANSI standard.

Table 7-1 Predefined Macros for ANSI Standard Conformance

Macro	Description
_ _DATE_ _	The date of compilation as a string literal in the form “Mmm dd yyyy”.
_ _FILE_ _	A string literal representing the name of the file being compiled.
_ _LINE_ _	The current line number as a decimal constant.
_ _STDC_ _	The constant 1 under ANSI conformance dialect (-Za); other dialects are 0.
_ _TIME_ _	The time of compilation, as a string literal in the form “hh:mm:ss”.

The compiler provides some predefined macros in addition to the predefined macros required by the ANSI standard. For a list of these macros see “Predefined Macros” in Chapter 6 of this manual and the section on the preprocessor category of the Microsoft *Visual C++ User’s Guide.*

Conformance to C++ Standards

The Intel C/C++ Compiler conforms to the Microsoft implementation of the C++ language. For more information on the Microsoft implementation of the C++ language, see the documentation for your copy of Microsoft Visual C++.

C++ Language Options

The Intel C/C++ Compiler lets you choose whether you want to enable C++ exception handling.

Enabling C++ Exception Handling (-GX, -GX-)

Use the -GX option to enable exception handling for C++. Exception handling is disabled by default.

To enable exception handling, use the following syntax:

```
prompt> icl -GX file.cpp
```

To disable exception handling for C++, use the -GX- option.

Run-time Type Information (-GR, -GR-)

Use the -GR option to allow the use of run-time type information features in your code. Use -GR- to disable run-time type information.

Microsoft Compatibility 8

The Intel C/C++ Compiler supports the Microsoft Visual C++ extensions to the C and C++ languages. This chapter describes the differences in the way the Intel C/C++ Compiler interprets some of the Microsoft Visual C++ extensions. The differences are in the following areas:

- Compiler Pragmas
- Microsoft compatibility option (`-Qms`)
- Unsupported Compiler Options
- Differences in PCH Support
- Compilation and Execution Differences

Compiler Pragmas

The Intel C/C++ Compiler supports the Microsoft Visual C++ pragmas with the following limitations:

- The pragma `pragma optimize` accepts a list that allows the enabling and disabling of specific optimizations. The Intel compiler ignores this list. The pragma either enables or disables all optimizations specified by options on the command line.
- The following pragmas are accepted without error but have no effect:

```
pragma auto_inlinepragma component
pragma function pragma include_alias
pragma inline_depthpragma inline_recursion
pragma intrinsicpragma setlocale
pragma warning
```

For more information on pragmas, see the on-line help or the documentation that accompanies your copy of Microsoft Visual C++.

Microsoft Compatibility Option (-Qms)

In a limited number of cases, the Microsoft compiler compiles source code without generating an error while the Intel compiler generates an error for the same source code. If this occurs in a file that is part of a third party distribution (such as include files or C++ class library files), it might not be convenient to correct the error in the source file. In such cases, using the `-Qms` option with the Intel compiler might allow a successful compilation.

Unsupported Compiler Options

The Intel C/C++ Compiler supports most of the same options as the Microsoft Visual C++ Compiler, as well as Intel-specific options. However, a small subset of Microsoft Visual C++ Compiler options are not supported by the Intel C/C++ Compiler. Most of the unsupported options, while useful for development purposes, are not required to build a working application. The unsupported options are listed in Table 8-1.

Table 8-1 List of Unsupported Microsoft Visual C++ Compiler Options

Option	Description
`-Fd`	name the PDB file used for debug information for specified source files
`-FR`	generate source browser information
`-Fr`	generate source browser information, excluding information on locals
`-GA`	optimize for Windows application
`-GD`	optimize for Windows DLL
`-Gi`	enable incremental compilation
`-Gm`	enable minimal rebuild

continued ☛

Table 8-1 List of Unsupported Microsoft Visual C++ Compiler Options

Option	Description
-Oa	assume no aliasing
-Ow	assume no cross-function aliasing
-WX	treat warnings as errors
-Yd	put debug information in every object (Microsoft PCH-specific option)
-Zg	generate function prototypes
-Zm	set compiler's memory allocation limit
-Zn	turn off packing of source browser information

The Intel C/C++ Compiler issues a remark stating lack of support for many of these options, but it silently ignores the following options: -Fd, -FR, -Fr, -Gi, -Gm, -Yd, -Zm, and -Zn.

Differences in PCH Support

There are some differences in how precompiled header (PCH) files are supported between the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler. These differences include the following:

- The PCH information generated by the Intel C/C++ Compiler is not compatible with the PCH information generated by the Microsoft Visual C++ Compiler.
- The Intel C/C++ Compiler does not support PCH generation and use in the same translation unit.
- The Intel C/C++ Compiler does not generate PCH information beyond a point where a declaration is seen in the primary translation unit. The Microsoft Visual C++ compiler generates PCH information beyond this point in some situations.
- The Intel C/C++ compiler validates the contents of a PCH file against the contents of a source file to make sure that they are semantically identical. If the PCH file and the source file up to the stop point are not semantically identical, the Intel C/C++ compiler does not use the PCH

file. In contrast, the Microsoft Visual C++ compiler does not perform any validation and uses the PCH file, even if it is syntactically and semantically different from the source.

Compilation and Execution Differences

While the Intel C/C++ Compiler is compatible with the Microsoft Visual C++ Compiler, there are some differences that can prevent successful compilation. Also there can be some incompatible generated-code behavior of some source files with the Intel C/C++ Compiler. In most cases, a modification of the user source file allows successful compilation with both the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler. These differences are listed as follows:

- The Intel C/C++ Compiler differs from the Microsoft Visual C++ Compiler in the way it expands preprocessor macros that are used within `#include` directives. Notice that in the following example, the two compilers behave similarly. When the code passes a macro as a parameter to another macro that uses the token-concatenation operator, the macro that is used as a parameter is not expanded before concatenation. This is demonstrated in the following example:

```
#define D   var
#define Decl(d)   int my ## d ## ;
Decl(D)
```

Both compilers would preprocess the preceding source and produce the following code:

```
int myD;
```

When a similar macro is used in a `#include` directive, the Intel C/C++ Compiler behaves similarly to the preceding example. However, the Microsoft Visual C++ Compiler performs an extra preprocessing scan through the `#include` directive to produce different results. In the following example, the `D` and `F` macros, when used in the `#include` directive, are not expanded by the Intel C/C++ Compiler.

```
#define  D    sys
#define  F    stat.h
```

```
#define INC(d,f)   < ## d ## / ## f ## >

#include INC(D,F)
// Visual C++ Compiler interpretation:
//# include <sys/stat.h>
// Intel C/C++ Compiler interpretation:
// #include <D/F>
```

The Intel C/C++ Compiler issues an error for this code when it fails to find the file called `D/F`. The Microsoft Visual C++ Compiler expands the `D` and `F` macros and accepts this code. The following alternative code achieves the same result but it works with both the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler:

```
#define  D    sys
#define  F    stat.h

#define CAT5(a,b,c,d,e)   a ## b ## c ## d ## e
#define INC(d,f)          CAT5(<,d,/,f,>)

#include INC(D,F)
```

- In C++ source, the Intel C/C++ Compiler does not treat a comma-expression that evaluates to zero as equivalent to a null pointer. The Microsoft Visual C++ Compiler treats such an expression as equivalent to a null pointer. Because of this difference, the following code will not compile with the Intel C/C++ Compiler:

```
struct C {
C(int,int,int,int);
C(C*);
};
void func()
{
C c = (0,0,0,0);// error:no suitable constructor
                // exists to convert from "int" to "C"
}
```

The Microsoft Visual C++ Compiler will compile this code, but because of the `=` operator, it will emit a call to `C(C*)` to convert from `0` (null pointer) to `C`. However, the programmer probably intended to call `C(int,int,int,int)`, which would be done with the following call:

```
C c(0,0,0,0);     // note: no '=' is used
```

- The Intel C/C++ Compiler differs from the Microsoft Visual C++ Compiler in the evaluation of left shift operations where the right operand, or shift count, is equal to or greater than the size of the left operand expressed in bits. The ANSI C Standard states that the behavior of such left-shift operations is undefined, meaning a program should not expect a certain behavior from these operations. This difference is only evident when both operands of the shift operation are constants. The following example illustrates this difference between the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler:

```
int x;
int y = 1;

void func()
{
 x = 1 << 32;
    // Visual C++ Compiler generates code to set x=0
    // Intel C/C++ Compiler generates code to set x=1
 y = y << 32;
    // Visual C++ Compiler generates code to set x=1
    // Intel C/C++ Compiler generates code to set x=1
}
```

Diagnostic Information 9

This chapter describes the various messages that the compiler produces. These messages include the sign-on message and diagnostic messages for remarks, warnings, or errors. The compiler always displays any error message, along with the erroneous source line, on the standard error device.

This chapter also describes how to control the severity of error messages.

Disabling the Sign-on Message (-nologo)

The compiler displays the following message each that you invoke it:

```
Intel C/C++ Compiler Version x.y.z ID
Copyright (C) years Intel Corporation.  All rights reserved.
```

ID	The unique identification number for this compiler.
x.y.z	Identifies the version of the compiler.
years	The years for which the software is copyrighted.

If you want to suppress the message, specify the `-nologo` option.

Printing the List of icl Options (-?, -help)

You can print a list of the most useful `icl` options by specifying the `-help` (or `-?`) option to the compiler. To print this list, use this command:

```
prompt> icl -help
```

or

```
prompt> icl -?
```

Diagnostic Messages

Diagnostic messages provide syntactic and semantic information about your source text. Syntactic information can include syntax errors and use of non-ANSI C. Semantic information includes unreachable code.

Diagnostic messages can be any of the following: command-line diagnostics, remark messages, warning messages, error messages, or catastrophic error messages.

Command-line diagnostics. These messages report improper command-line options or arguments. If the command line contains an unrecognized option, the compiler passes the option to the linker. If the linker still does not recognize the option, the linker produces the diagnostic message.

Command-line error messages appear on the standard error device in the form:

`icl: Command line error: message`

message	Describes the error.

Command-line warning messages appear as follows:

`icl: Command line warning: message`

Language Diagnostics. These messages describe diagnostics that are reported during the processing of the source file. These diagnostics have the following format:

`filename(linenum): type nn: message`

filename	Indicates the name of the source file currently being processed.
linenum	Indicates the source line where the compiler detects the condition.
type	Indicates the severity of the diagnostic message: `warning`, `remark`, `error`, or `catastrophic error`.
nn	The number assigned to the `error` (or `warning`) message.
message	Describes the diagnostic.

The following is an example of a warning message:

```
tantst.cpp(3): warning 328: Local variable “increment”
never used.
```

The compiler can also display internal error messages on the standard error device. If your compilation produces any internal errors, contact your Intel representative. Internal error messages are in the following form:

```
FATAL COMPILER ERROR: message
```

Remark messages. These messages report common but sometimes unconventional use of C or C++. The compiler does not print or display remarks unless you specify level 4 for the `-W` option, as described in “Suppressing Warning Messages or Enabling Remarks (-w, -Wn)” later in this chapter. Remarks do not stop translation or linking. Remarks do not interfere with any output files. The following are some representative remark messages:

```
function declared implicitly
type qualifiers are meaningless in this declaration
controlling expression is constant
```

Warning Messages. These messages report legal but questionable use of the language in the program being compiled. The compiler displays warnings by default. You can suppress warning messages by setting the diagnostic level to `0`. Warnings do not stop translation or linking. Warnings do not interfere with any output files. The following are some representative warning messages:

```
declaration does not declare anything
pointless comparison of unsigned integer with zero
possible use of = where == was intended
```

Error messages. These messages report syntactic or semantic misuse of C or C++. The compiler always displays error messages. Errors suppress object code for the module containing the error and prevent linking, but they allow parsing to continue to scan for any other errors. Some representative error messages are:

```
missing closing quote
expression must have arithmetic type
expected a “;”
```

Catastrophic errors. These messages indicate environmental problems. Catastrophic error conditions stop translation, assembly, and linking. If a catastrophic error ends compilation, the compiler displays a termination message on standard error output. The following are some representative catastrophic error messages:

```
out of memory
could not open temporary file filename
could not open source file filename
```

Suppressing Warning Messages with lint Comments

The `lint` program attempts to detect features of a C or C++ program that are likely to be bugs, non-portable, or wasteful. The compiler recognizes three `lint`-specific comments: `/*ARGSUSED*/`, `/*NOTREACHED*/`, and `/*VARARGS*/`. Like the UNIX `lint` program, the compiler suppresses warnings about certain conditions when you place these comments at specific points in the source.

Suppressing Warning Messages or Enabling Remarks (-w, -W*n*)

Use the `-w` or `-Wn` option to suppress warning messages or to enable remarks during the preprocessing and compilation phases. You can enter the option with one of the following arguments:

`-W0,-w`	Displays error messages only.
`-W1,-W2,-W3`	Displays warnings and error messages. The compiler uses this level as the default.
`-W4`	Displays remarks, warnings, and error messages.

For some compilations, you might not want warnings for known and benign characteristics, such as the K&R C constructs in your code. For example, the following command compiles `newprog.cpp` and displays compiler errors, but not warnings:

```
prompt> icl -W0 newprog.cpp
```

Controlling the Severity of Diagnostics (-Qwd, -Qwr, -Qww, -Qwe)

You can control the severity of some of the diagnostics returned by the compiler. The compiler returns two types of diagnostics:

hard errors	Diagnostics issued for code that is definitely wrong or questionable. The severity of a hard error is not configurable. For hard errors, the message number is never printed. Remarks and warnings are never considered hard errors.
soft diagnostics	All other diagnostics (including remarks and warnings). For soft diagnostics, the message number is always printed. The severity of a soft diagnostic is configurable by the options described below.

In the descriptions below, *tag* represents the number associated with the diagnostic.

`-Qwd`*tag*	Disable the soft diagnostic that corresponds to *tag*.
`-Qwr`*tag*	Override the severity of the soft diagnostic corresponding to *tag* and make it a remark.
`-Qww`*tag*	Override the severity of the soft diagnostic corresponding to *tag* and make it a warning.
`-Qwe`*tag*	Override the severity of the soft diagnostic corresponding to *tag* and make it an error.

For example, the following command line disables soft diagnostic `68` during compilation of the file `a.cpp`:

```
prompt> icl -Qwd68 -c a.cpp
```

The following command line changes the severity of soft diagnostics `68` and `152` to remarks during compilation of the file `a.cpp`.

```
prompt> icl -Qwr68,152 -c a.cpp
```

Example. Assume that you have a file `x.cpp` that contains the following lines:

```
/*
/* This is a comment
*/

extern i;
```

If you compile this code with warnings enabled (the default), you will receive the following response from the compiler:

```
x.cpp(2): warning 9: nested comment is not allowed
   /* This is a comment
   ^
x.cpp(5): warning 260: explicit type is missing ("int"
assumed)
   extern i;
   ^
```

If you compile the code with the option `-Qwd9`, (to disable warning number 9), you will receive the following response from the compiler:

```
x.cpp(5): warning 260: explicit type is missing ("int"
assumed)
   extern i;
   ^
```

Limiting the Number of Errors Reported (-Qwn)

Use the `-Qwn` option to set the number of error messages displayed before the compiler aborts. By default, if more than 100 errors are displayed, compilation aborts.

`-Qwn`*num* Limit the number of error diagnostics that will be displayed prior to aborting compilation to *num*. Remarks and warnings do not count towards this limit.

For example, the following command line specifies that if more than 50 error messages are displayed during the compilation of `a.cpp`, compilation aborts.

```
prompt> icl -Qwn50 -c a.cpp
```

Additional Information about the Compilation

The compilation system issues a variety of self-explanatory information messages about the compilation system components and the process that are not described in this manual.

Libraries 10

The compiler allows you to use all the standard run-time libraries that are part of Microsoft Visual C++ Version 4.0 or later. You can determine the libraries used by your applications by controlling the linker or by using the options described in this chapter.

Managing Libraries

The `LIB` environment variable contains a semicolon-separated list of directories in which the Microsoft linker will search for library (`.lib`) files. If you want the linker to search additional libraries, you can add their names to the command line, to a response file, or to the `icl.cfg` file. In each case, the names of these libraries are passed to the linker before the names of the Intel libraries (`libm.lib` or `libm_chk.lib`) and the Microsoft-provided default libraries that the driver always specifies (`oldnames.lib` and `libc.lib`). For more information on adding library names to the response file and the configuration file (`icl.cfg`), see "Response Files" and "Configuration Files" in Chapter 3.

To specify a library name on the command line, you must first add the library's path to the `LIB` environment variable. Then, to compile `file.cpp` and link it with the library `mylib.lib` enter the following command:

```
prompt> icl file.cpp mylib.lib
```

The compiler driver, `icl.exe`, passes file names to the Microsoft linker in the following order:

1. The object file
2. Any objects or libraries specified on the command line, in a response file, or in a configuration file
3. The `libm.lib` library (or, if you specified the `-QIfdiv` option, the `libm_chk.lib` libraries)
4. The Microsoft-provided libraries `libc.lib` and `oldnames.lib`

Default Libraries

The compiler allows you to use all the standard run-time libraries that are part of Microsoft Visual C++ Version 4.x. By default, the compiler automatically expands a number of standard C, C++, and math library functions. For more information, see "In-line Expansion of Library Functions (-Oi, -Oi-)" in Chapter 4.

Library Files

The compiler automatically tells the linker to use the following support libraries.

`libm_chk.lib` or `libm.lib`	The `libm.lib` file contains the math library. The `libm_chk.lib` file contains the math library with support routines for a floating-point division software patch for certain steppings of the Pentium processor. For information on these libraries, see "Enabling the Floating-Point Division Check (-QIfdiv)".

These libraries are supplied with Microsoft Visual C++ Version 4.0 or later.

`libc.lib`	The standard C run-time library.
`oldnames.lib`	Contains aliases for non-ANSI functions that normally begin with an underscore in Microsoft libraries.

If you want to link your program with the alternate or additional libraries, specify them at the end of your `icl` command line. For example, to compile and link `hello.cpp` with `mylib.lib`, use the following command:

```
prompt> icl -Fehello.exe hello.cpp mylib.lib
```

The `mylib.lib` library appears prior to the `libc.lib` library in the command line for the `LINK` linker.

Math Libraries

In the compiler package, you received the Intel math library, `libm.lib`, which contains optimized versions of the math functions in the standard C run-time library. The functions in the library are optimized for program execution speed on the Pentium processor.

To use the optimized math library, you must place `libm.lib` in one of the directories specified in the library search path defined by the `LIB` variable. Intel recommends that you place `libm.lib` in the first directory specified in the path.

Enabling the Floating-Point Division Check (-QIfdiv)

The `-QIfdiv` option enables a software patch for the floating-point division flaw that exists on some steppings of the Pentium processor. This patch ensures that the precision of your floating-point division calculations are correct.

NOTE. *This option (`-QIfdiv`) is no longer enabled by default when you specify either the `-G3` or the `-G5` options.*

When the `-QIfdiv` option is enabled, the compiler uses `libm_chk.lib`, which is a special version of the Intel-supplied library, to link your programs. This file is linked automatically. The `libm_chk.lib` library contains the support routines for the floating-point division software patch and checked versions of the affected math library functions.

The `-QIfdiv-` option disables the software patch for the floating-point division flaw regardless of whatever other options are specified. When you specify `-QIfdiv-`, the compiler uses simple hardware instructions for floating-point division and affected intrinsics. If you specify the `-QIfdiv-`

option, the compiler does not need a special version of the Intel-supplied optimized math library to link your programs. Similarly, if you choose not to use the special version of the optimized math library, you must specify `-QIfdiv-`. This option is the default if you specify either the `-G4` or the `-G6` options.

Avoiding Incorrect Decoding of Certain Instructions (-QI0f)

Some instructions have 2-byte opcodes, the first byte of which contains 0f. In rare cases, the Pentium processor incorrectly decodes these instructions. Specify the `-QI0f` option to avoid the incorrect decoding of these instructions. The work-around implemented in the Intel C/C++ Compiler avoids generating the susceptible instructions.

Controlling Compiler-Generated Code 11

This chapter describes options and features that let you control the outcome of Intel compiler-generated code without interfering with the way your program runs.

- "Specifying Structure Tag Alignments (-Zp)" describes how you can set the alignment of structures and unions from the command line instead of adding a pragma to your source file.
- "Allocating Memory for Block Variables (-Qscope_alloc)" describes how you can allocate the minimum stack space needed for block variables in nested scopes instead of allowing the compiler to allocate the stack space.
- "Allocation of Zero-initialized Variables (-Qnobss_init)" describes how you can place variables initialized to zero in the `DATA` section instead of allowing the compiler to place them in the `BSS` section.

Specifying Structure Tag Alignments (-Zp)

You can specify an alignment constraint for structures and unions in two ways: place a pack pragma in your source file or enter the alignment option on the command line. Both specifications change structure tag alignment constraints.

Use the `-Zp` option to determine the alignment constraint for structure declarations. Generally, smaller constraints result in smaller data sections while larger constraints support faster execution.

The form of this option is:

`-Zp`*`number`*

`number`	The alignment constraint indicated by one of the following values:
`1`	1 byte.
`2`	2 bytes.
`4`	4 bytes.
`8`	8 bytes.
`16`	16 bytes.

For example, to specify 2 bytes as the alignment constraint for all structures and unions in the file `prog1.e`, use the following command:

```
prompt> icl -Zp2 prog1.e
```

Allocating Memory for Block Variables (-Qscope_alloc)

Use the `-Qscope_alloc` option to allocate only the minimum stack space needed for any nested scope. This option is useful if you have limited stack space or if your memory usage is critical. By default, the compiler allocates the sum of all block automatic (local) variables in nested scopes.

Allocation of Zero-initialized Variables (-Qnobss_init)

Use the `-Qnobss_init` option to place any variables that are explicitly initialized with zeros in the `_DATA` section. By default, variables explicitly initialized with zeros are placed in the `_BSS` section, but some programs require them to be in the `_DATA` section.

MMX™ Technology Intrinsics

12

The Pentium Pro and Pentium processors with MMX technology include extensions for multimedia programming. The compiler supports the use of these MMX instructions in C programs by the use of intrinsics. These intrinsics are coded with the syntax of C function calls, but cause the compiler to generate the corresponding MMX instructions. The compiler allows you to use C variables in place of hardware registers. This frees you from the management of these registers. The compiler also schedules the instructions to maximize performance.

The MMX technology intrinsics operate on a 64-bit data type. Typically, they have two arguments and one result, that corresponds to the two source operands and one destination of the hardware instructions. To use the MMX technology intrinsics, you must include the file `mmintrin.h` found in the Intel compiler include directory. A 64-bit data type `__m64` is defined in this include file. The include file also contains the ANSI C prototypes for these intrinsics.

Intel recommends that floating-point instructions not be used in a routine containing MMX technology intrinsics. There is a hardware delay when floating-point instructions are mixed with MMX instructions.

The following sections describe each of the MMX technology intrinsics in the form of an ANSI C prototype followed by a brief description. For complete details of the hardware instructions see the *Intel Architecture MMX Technology Programmer's Reference Manual,* order number *243007*.

General Support Intrinsics

`void _m_empty (void)`

Empty the multimedia state.

`__m64 _m_from_int (int i)`

Convert the integer object *i* to a 64-bit `__m64` object. The integer value is zero extended to 64 bits.

`int _m_to_int (__m64 m)`

Convert the lower 32 bits of the `__m64` object *m* to an integer.

`__m64 _m_packsswb (__m64 m1, __m64 m2)`

Pack the eight 16-bit values found in *m1* and *m2* into eight 8-bit values with signed saturation.

`__m64 _m_packssdw (__m64 m1, __m64 m2)`

Pack the four 32-bit values found in *m1* and *m2* into four 16-bit values with signed saturation.

`__m64 _m_packuswb (__m64 m1, __m64 m2)`

Pack the eight 16-bit values found in *m1* and *m2* into eight 8-bit values with unsigned saturation.

`__m64 _m_punpckhbw (__m64 m1, __m64 m2)`

Interleave the four 8-bit values from the high half of *m1* with the four 8-bit values from the high half of *m2*. The result is four 16-bit values.

`__m64 _m_punpckhwd (__m64 m1, __m64 m2)`

Interleave the two 16-bit values from the high half of *m1* with the two 16-bit values from the high half of *m2*. The result is two 32-bit values.

`__m64 _m_punpckhdq (__m64 m1, __m64 m2)`

Interleave the 32-bit value from the high half of *m1* with the 32-bit value from the high half of *m2*. The result is a 64-bit value.

`__m64 _m_punpcklbw (__m64 m1, __m64 m2)`

Interleave the four 8-bit values from the low half of *m1* with the four 8-bit values from the low half of *m2*. The result is four 16-bit values.

`__m64 _m_punpcklwd (__m64 m1, __m64 m2)`

Interleave the two 16-bit values from the low half of *m1* with the two 16-bit values from the low half of m2. The result is two 32-bit values.

`__m64 _m_punpckldq (__m64 m1, __m64 m2)`

Interleave the 32-bit value from the low half of *m1* with the 32-bit value from the low half of *m2*. The result is a 64-bit value.

Packed Arithmetic Intrinsics

`__m64 _m_paddb (__m64 m1, __m64 m2)`

Add the eight 8-bit values in *m1* to the eight 8-bit values in *m2*.

`__m64 _m_paddw (__m64 m1, __m64 m2)`

Add the four 16-bit values in *m1* to the four 16-bit values in *m2*.

`__m64 _m_paddd (__m64 m1, __m64 m2)`

Add the two 32-bit values in *m1* to the two 32-bit values in *m2*.

`__m64 _m_paddsb (__m64 m1, __m64 m2)`

Add the eight signed 8-bit values in *m1* to the eight signed 8-bit values in *m2* and saturate.

`__m64 _m_paddsw (__m64 m1, __m64 m2)`

Add the four signed 16-bit values in *m1* to the four signed 16-bit values in *m2* and saturate.

`__m64 _m_paddusb (__m64 m1, __m64 m2)`

Add the eight unsigned 8-bit values in *m1* to the eight unsigned 8-bit values in *m2* and saturate.

`__m64 _m_paddusw (__m64 m1, __m64 m2)`

Add the four unsigned 16-bit values in *m1* to the four unsigned 16-bit values in *m2* and saturate.

`__m64 _m_psubb (__m64 m1, __m64 m2)`

Subtract the eight 8-bit values in *m2* from the eight 8-bit values in *m1*.

`__m64 _m_psubw (__m64 m1, __m64 m2)`

Subtract the four 16-bit values in *m2* from the four 16-bit values in *m1*.

`__m64 _m_psubd (__m64 m1, __m64 m2)`

Subtract the two 32-bit values in *m2* from the two 32-bit values in *m1*.

`__m64 _m_psubsb (__m64 m1, __m64 m2)`

Subtract the eight signed 8-bit values in *m2* from the eight signed 8-bit values in *m1* and saturate.

`__m64 _m_psubsw (__m64 m1, __m64 m2)`

Subtract the four signed 16-bit values in *m2* from the four signed 16-bit values in *m1* and saturate.

`__m64 _m_psubusb (__m64 m1, __m64 m2)`

Subtract the eight unsigned 8-bit values in *m2* from the eight unsigned 8-bit values in *m1* and saturate.

`__m64 _m_psubusw (__m64 m1, __m64 m2)`

Subtract the four unsigned 16-bit values in *m2* from the four unsigned 16-bit values in *m1* and saturate.

`__m64 _m_pmaddwd (__m64 m1, __m64 m2)`

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.

`__m64 _m_pmulhw (__m64 m1, __m64 m2)`

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* and produce the high 16 bits of the four results.

`__m64 _m_pmullw (__m64 m1, __m64 m2)`

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* and produce the low 16 bits of the four results.

Shift Intrinsics

`__m64 _m_psllw (__m64 m, __m64 count)`

Shift four 16-bit values in *m* left the amount specified by *count* while shifting in zeros.

`__m64 _m_psllwi (__m64 m, int count)`

Shift four 16-bit values in *m* left the amount specified by *count* while shifting in zeros. For the best performance, count should be a constant.

`__m64 _m_pslld (__m64 m, __m64 count)`

Shift two 32-bit values in *m* left the amount specified by *count* while shifting in zeros.

`__m64 _m_pslldi (__m64 m, int count)`

Shift two 32-bit values in *m* left the amount specified by *count* while shifting in zeros. For the best performance, count should be a constant.

`__m64 _m_psllq (__m64 m, __m64 count)`

Shift the 64-bit value in *m* left the amount specified by *count* while shifting in zeros.

`__m64 _m_psllqi (__m64 m, int count)`

Shift the 64-bit value in *m* left the amount specified by *count* while shifting in zeros. For the best performance, count should be a constant.

`__m64 _m_psraw (__m64 m, __m64 count)`

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in the sign bit.

`__m64 _m_psrawi (__m64 m, int count)`

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in the sign bit. For the best performance, count should be a constant.

`__m64 _m_psrad (__m64 m, __m64 count)`

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in the sign bit.

`__m64 _m_psradi (__m64 m, int count)`

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in the sign bit. For the best performance, count should be a constant.

`__m64 _m_psrlw (__m64 m, __m64 count)`

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in zeros.

`__m64 _m_psrlwi (__m64 m, int count)`

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in zeros. For the best performance, count should be a constant.

`__m64 _m_psrld (__m64 m, __m64 count)`

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in zeros.

`__m64 _m_psrldi (__m64 m, int count)`

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in zeros. For the best performance, count should be a constant.

`__m64 _m_psrlq (__m64 m, __m64 count)`

Shift the 64-bit value in *m* right the amount specified by *count* while shifting in zeros.

`__m64 _m_psrlqi (__m64 m, int count)`

Shift the 64-bit value in *m* right the amount specified by *count* while shifting in zeros. For the best performance, count should be a constant.

Logical Intrinsics

`__m64 _m_pand (__m64 m1, __m64 m2)`

Perform a bitwise AND of the 64-bit value in *m1* with the 64-bit value in *m2*.

`__m64 _m_pandn (__m64 m1, __m64 m2)`

Perform a logical NOT on the 64-bit value in *m1* and use the result in a bitwise AND with the 64-bit value in *m2*.

`__m64 _m_por (__m64 m1, __m64 m2)`

Perform a bitwise OR of the 64-bit value in *m1* with the 64-bit value in *m2*.

`__m64 _m_pxor (__m64 m1, __m64 m2)`

Perform a bitwise XOR of the 64-bit value in *m1* with the 64-bit value in *m2*.

Compare Intrinsics

`__m64 _m_pcmpeqb (__m64 m1, __m64 m2)`

If the respective 8-bit values in *m1* are equal to the respective 8-bit values in *m2* set the respective 8-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpeqw (__m64 m1, __m64 m2)`

If the respective 16-bit values in *m1* are equal to the respective 16-bit values in *m2* set the respective 16-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpeqd (__m64 m1, __m64 m2)`

If the respective 32-bit values in *m1* are equal to the respective 32-bit values in *m2* set the respective 32-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpgtb (__m64 m1, __m64 m2)`

If the respective 8-bit values in *m1* are greater than the respective 8-bit values in *m2* set the respective 8-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpgtw (__m64 m1, __m64 m2)`

If the respective 16-bit values in *m1* are greater than the respective 16-bit values in *m2* set the respective 16-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _m_pcmpgtd (__m64 m1, __m64 m2)`

If the respective 32-bit values in *m1* are greater than the respective 32-bit values in *m2* set the respective 32-bit resulting values to all ones, otherwise set them all to zeros.

Compiler Limits

A

Table A-1 shows the size or number of each item that the compiler can process. All capacities shown in the table are tested values; the actual number can be greater than the number shown.

Table A-1 Compiler Limits

Item	Tested Values
Control structure nesting (block nesting)	128
Conditional compilation nesting	32
Declarator modifiers	32
Parenthesis nesting levels	128
Significant characters, internal identifier	128
External identifier name length	33
Number of identifiers in a single block	512
Number of macros simultaneously defined	4096
Number of parameters to a function call	128
Number of characters in logical line	4096
Number of characters in a string	4096
Bytes in an object	512 million
Include file nesting depth	32
Case labels in a switch	1024
Members in one structure or union	512

continued ☛

Table A-1 Compiler Limits (continued)

Item	Tested Values
Enumeration constants in one enumeration	512
Levels of structure nesting	64
Number of external identifiers/files	2048
Number of parameters per macro	128

Experimental Performance Tuning

B

This appendix provides information on how you can adjust the compiler's optimization for a particular application by experimenting with memory and interprocedural optimizations.

NOTE. *The suboptions described in this chapter are provided for the benefit of developers who are willing to experiment with advanced optimizations. Please be aware that experimenting with these advanced optimization can yield unexpected results. Therefore, this guide describes only the basic functionality of these advanced optimizations.*

Keywords for Optimization (-QW0)

Enter the `-QW0` option with the applicable keywords to select particular in-line expansions and loop optimizations. Enter the option with a `-Qmem` or `-Qip` specification, as follows:

`{-Qmem | -Qip}  [-QW0,-`*`keyword`*`[,-`*`keyword`*`]] ...`

`keyword` is any of the applicable optimizations for the specified level.

You can also simultaneously refine memory and interprocedural optimizations by placing keywords for both options in one `-QW0` entry. The compiler performs interprocedural optimizations before performing memory-access optimizations.

Memory Optimization

If you enter the `-Qmem` option without keywords, the compiler performs only loop interchange optimization (the default action). The keywords for `-Qmem` are the following:

`mo_interchange=FALSE`	Disables loop interchange.
`mo_alternate_loops`	Creates alternate loops when loop bounds are not known.
`mo_block_size=`*`nn`*	Specifies the size of loop blocks.
`mo_block_loops`	Places loop blocks in cache memory to reduce the distance between loop iterations. You can improve your program's cache usage by combining this option with, or substituting it for, the `mo_block_size=`*`nn`* option. Such usage balances memory access and loop computation.
`mo_strip_mine`	Creates an additional level of nesting so that computations of loop vectors remain in cache memory. This optimization is also known as strip-mining.
`mo_distribute`	Distributes statements in a loop body to reduce the potential for time-consuming bus traffic due to concentrated memory accesses.
`mo_siblings_limit=`*`n`*	Specifies the number of loops in a nested loop before optimization stops.

The following command performs procedural optimizations, loop-blocking and loop-distributing, in addition to loop interchange, on the `factor.cpp` source file:

```
prompt> icl -Qmem -QW0,-mo_block_loops,-mo_distribute
        factor.cpp
```

The compiler automatically performs loop interchange optimization with memory optimizations.

NOTE. *Activating these memory keywords can increase compilation time substantially. Use them for programs with many nested loops that access data in regular patterns. For such programs, interprocedural optimization combined with memory optimization yields greater performance, as described earlier in this appendix.*

The performance of the generated code can vary drastically with the specific hardware configuration you use. Performance is affected by such factors as the presence or absence of a second-level cache and the particular memory configuration of your system.

Interprocedural Optimization

If you specify the `-Qip` option without the `-QW0` qualification, the compiler expands functions in line, propagates constant arguments, passes arguments in registers, and monitors module-level static variables. Use the following `-QW0` keywords to refine these interprocedural optimizations:

`ip_args_in_regs=0` — Disables the passing of arguments in registers. By default, external functions can pass arguments in registers when called locally. Normally, only static functions can pass arguments in registers, provided the address of the function is not taken and the function does not use a variable number of arguments.

`ip_inline_max_calls=N`	Used with the inline heuristics (`-Qinl_heur`) option. This option changes the default number of call-sites to in-line. Note that *N* call-sites are in-lined only if that many call-sites meet the minimum in-line criteria. For more information, see the "Using In-line Heuristics (-Qinl_heur n)" and "Criteria for In-Line Function Expansion" sections.
`ip_inline_max_stats=n`	Sets the allowable number of intermediate language statements for a function that is expanded in line. The number *n* is a positive integer. The number of intermediate language statements usually exceeds the actual number of source language statements. The default is set to the maximum number of 200.
`ip_inline_max_total_stats=n`	Sets the maximum increase in the number of intermediate language statements for each function that is expanded in line. The number *n* is a positive integer. By default, each function can increase to a maximum of 5000 statements.
`ip_no_external_ref`	Indicates that the source file contains the main program and does not contain functions that are referenced by external functions. If you do not specify this option, the compiler retains an original copy of each expanded in-line function.

Analyzing the Effects of Multifile IPO (-Qipo_c, -Qipo_S)

The `-Qipo_c` and `-Qipo_S` options are useful for analyzing the effects of multifile IPO, or when experimenting with multifile IPO between modules that do not make up a complete program.

Use the `-Qipo_c` option to optimize across files and produce an object file. This option performs optimizations as described for `-Qipo`, but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.obj`. You can use the `-Fe` option to specify a different name as shown in the following example:

```
prompt> icl -G6 -Qipo_c -Fefilename a.cpp b.cpp c.cpp
```

Use the `-Qipo_S` option to optimize across files and produce an assembly file. This option performs optimizations as described for `-Qipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.asm`. You can use the `-Fe` option to specify a different name as shown in the following example:

```
prompt> icl -G6 -Qipo_S -Fefilename a.cpp b.cpp c.cpp
```

Using In-line Heuristics (-Qinl_heur *n*)

Use the `-Qinl_heur` *n* option to in-line the most frequently used call-sites. When you specify this option, call-sites are treated as leaf routines. For more information on in-lining and the minimum in-lining criteria, see "In-line Function Expansion" in Chapter 4.

The `-Qinl_heur` *n* option only takes effect when used together with the `-Qip` or `-Qipo` option. The value of *n* can be `0`, `1`, `2`, or `3`.

NOTE. *In-line heuristics* `1` *through* `3` *are all profile-guided. If no profile information is available the compiler will use the default in-line heuristics (that is,* *n* = `0`*) and issue a warning. See* *"Profile-Guided Optimization" in Chapter 4* *to find out how to create profile information.*

There are four variations on the `-Qinl_heur n` option:

`-Qinl_heur 0`	Default in-line heuristics. These heuristics work both with or without available profile information.
`-Qinl_heur 1`	In-line the *N* most frequently executed call-sites.
`-Qinl_heur 2`	In-line the *N* most frequently executed call-sites. After each in-lining, the execution counts are scaled and the list of frequently executed call-sites is updated.
`-Qinl_heur 3`	In-line the *N* most frequently executed call-sites scaled by the size of the callee. After each in-lining, the execution counts are scaled and the list of call-sites updated.

By default, *N* = 100. You can change this value by using the option `-QW0,-ip_inline_max_calls=N`. Note that *N* call-sites will be in-lined only if that many call-sites meet the minimum in-line criteria.

Criteria for In-Line Function Expansion

For a routine to be considered for in-lining, it has to meet certain minimum criteria. There are criteria to be met by the call-site, the caller, and the callee. The call-site is the site of the call to the function that might be in-lined. The caller is the function that contains the call-site. The callee is the function being called that might be in-lined.

Minimum call-site criteria:

- The number of actual arguments must match the number of formal arguments of the callee.
- The number of return values must match the number of return values of the callee.
- The data types of the actual and formal arguments must be compatible.
- No multi-lingual in-lining is allowed. Caller and callee must be written in the same source language.

Minimum criteria for the caller:

- At most 2000 intermediate statements will be in-lined into the caller from all the call-sites being in-lined into the caller. You can change this value by specifying the option `-QW0,-ip_inline_max_total_stats=` *`new value`*
- Does not contain `_ _asm` insertions.
- The function must be called if it is declared as `static`. Otherwise, it will be deleted.

Minimum criteria for the callee:

- Does not have variable argument list.
- Is not considered infrequent due to the name. Routines that contain the following substrings in their names are not in-lined: `abort`, `alloca`, `denied`, `err`, `exit`, `fail`, `fatal`, `fault`, `halt`, `init`, `interrupt`, `invalid`, `quit`, `rare`, `stop`, `timeout`, `trace`, `trap`, and `warn`.
- Does not contain `asm` statements.
- Is not considered unsafe for other reasons.

Once these criteria are met, the compiler picks the routines whose in-line expansions provide the greatest benefit to program performance. This is done using the following default heuristics. When you use profile-guided optimizations, a number of other heuristics are used (see "Profile-Guided Optimization" in Chapter 4 for more information on profile-guided optimization and "Using In-line Heuristics (-Qinl_heur n)" above for more information on heuristics).

- The default heuristic focuses on call-sites in loops or calls to functions containing loops.
- When profile information is available, the focus changes to the most frequently executed call-sites.

  Also, the default in-line heuristic does not allow the in-lining of functions with more than 400 intermediate statements, or the number specified by the option `-W0,-ip_inline_max_stats`.
- The default in-line heuristic stops in-lining when direct recursion is detected.

- The default heuristic will always in-line very small functions that meet the minimum in-line criteria. By default, functions with 10 or fewer intermediate statements are in-lined. This limit can be modified with the option `-W0,-ip_inline_min_stats`.

Glossary

alignment constraint	The proper boundary of the stack where data must be stored.
alternate loop transformation	An optimization in which the compiler generates a copy of a loop and executes the new loop depending on the boundary size.
branch count profiler	A utility that counts the number of times a program executes each branch statement. The utility also generates a database that shows how the program executed.
branch probability database	The database generated by the branch count profiler. The database contains the number of times each branch is executed.
cache hit	An instance of the processor retrieving data from the cache.
call site	A call site consists of the instructions immediately preceding a call instruction and the call instruction itself.
common subexpression elimination	An optimization in which the compiler detects and combines redundant computations.
conditionals	Any operation that takes place depending on whether a certain condition is true.

constant argument propagation	An optimization in which the compiler replaces the formal arguments of a routine with actual constant values. The compiler then propagates constant variables used as actual arguments.
constant branches	Conditionals that always take the same branch.
constant folding	An optimization in which the compiler, instead of storing the numbers and operators for computation when the program executes, evaluates the constant expression and uses the result.
copy propagation	An optimization in which the compiler eliminates unnecessary assignments by using the value assigned to a variable instead of using the variable itself.
dataflow	The movement of data through a system, from entry to destination.
dead-code elimination	An optimization in which the compiler eliminates any code that generates unused values or any code that will never be executed in the program.
dynamic linking	The process in which a shared object is mapped into the virtual address space of your program at run time.
empty declaration	A semicolon and nothing before it.
frame pointer	A pointer that holds a base address for the current stack and is used to access the stack frame.
in-line function expansion	An optimization in which the compiler replaces each function call with the function body expanded in place.
induction variable simplification	An optimization in which the compiler reduces the complexity of an array index calculation by using only additions.
instruction scheduling	An optimization in which the compiler reorders the generated machine instructions so that more than one can execute in parallel.

instruction sequencing	An optimization in which the compiler eliminates less efficient instructions and replaces them with instruction sequences that take advantage of a particular processor's features.
interprocedural optimization	An optimization that applies to the entire program except for library routines.
loop blocking	An optimization in which the compiler reorders the execution sequence of instructions so that the compiler can execute iterations from outer loops before completing all the iterations of the inner loop.
loop unrolling	An optimization in which the compiler duplicates the executed statements inside a loop to reduce the number of loop iterations.
loop-invariant code movement	An optimization in which the compiler detects a computation that does not change within a loop and then moves the computation out of the loop.
MMX intrinsic	A function call that the compiler expands into one or more MMX instructions.
padding	The addition of bytes or words at the end of each data type in order to meet size and alignment constraints.
preloading	An optimization in which the compiler loads the vectors, one cache at a time, so that during the loop computation the number of external bus turnarounds is reduced.
profiling	A process in which detailed information is produced about the program's execution.
register variable detection	An optimization in which the compiler detects the variables that never need to be stored in memory and places them in register variables.
side effects	Results of the optimization process that might increase the code size and/or processing time.

static linking	The process in which a copy of the object file that contains a function used in your program is incorporated in your executable file at link time.
strength reduction	An optimization in which the compiler reduces the complexity of an array index calculation by using only additions.
strip mining	An optimization in which the compiler creates an additional level of nesting to enable inner loop computations on vectors that can be held in the cache. This optimization reduces the size of inner loops so that the amount of data required for the inner loop can fit the cache size.
token pasting	The process in which the compiler treats two tokens separated by a comment as one (for example, `a##b` becomes `ab`).
transformation	A rearrangement of code. In contrast, an optimization is a rearrangement of code where improved run-time performance is guaranteed.
unreachable code	Instructions that the compiler determines can never be executed.
unused code	Instructions that produce results that are not used in the program.
variable renaming	An optimization in which the compiler renames instances of a variable that refer to distinct entities.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z